B-Prolog User's Manual

(Version 6.4)

Prolog, Agent, and Constraint Programming

By Neng-Fa Zhou
Afany Software & CUNY Brooklyn & Kyutech

March 2003

Preface

Welcome to B-Prolog, a versatile and efficient constraint logic programming (CLP) system! B-Prolog is being brought to you by Afany Software (New York), KLS (Tokyo), and NandaSoft (Nanjing).

The birth of CLP is a milestone in the history of programming languages. CLP combines two declarative programming paradigms: logic programming and constraint solving. The declarative nature has proven appealing in numerous applications including computer-aided design and verification, database, software engineering, optimization, configuration, graphical user interface, and language processing. It greatly enhances the productivity of software development and software maintainability. In addition, because of the availability of efficient constraint-solving, memory management, and compilation techniques, CLP programs can be more efficient than their counterparts written in procedural languages.

B-Prolog is a Prolog system with extensions for programming concurrency, constraints, and interactive graphics. The system is based on a significantly refined WAM, called ATOAM[1], that facilitates software emulation. In addition to an ATOAM emulator with a garbage collector written in C, the system consists of a compiler and an interpreter written in Prolog, and a library of built-in predicates written in C and Prolog. B-Prolog accepts not only standard form Prolog programs but also *matching clauses* in which the determinacy and input/output unifications are denoted explicitly. Matching clauses are compiled into more compact and faster code than standard-form clauses. The compiler and most of the libraries are written in matching clauses.

B-Prolog follows the standard of Prolog but also enjoys several features that are not available in traditional Prolog systems. B-Prolog provides an interactive environment through which users can consult, list, compile, load, debug and run programs. The command editor in the environment facilitates recalling and editing old commands. B-Prolog provides a bi-directional interface with C and Java. This interface makes it possible to mix Prolog with C, C++, and Java. B-Prolog offers the user a unique construct, called *action rules*, which is useful for programming concurrency, implementing constraint propagators, and developing interactive user interfaces. Action rules have been successfully used to implement constraint solvers over trees, Boolean, finite-domains, and sets. B-Prolog also provides a high-level and constraint-based graphics library, called *CGLIB*. The library includes primitives for creating and manipulating graphical objects and a set of constraints that facilitates the specification of the layouts of objects. Action rules are used to program interactions.

This document describes the B-Prolog system and its usage. It consists of the following three parts.

Part-I Prolog Programming

[1]N.F. Zhou: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Transactions on Programming Languages and Systems*, Vol18, No6, pp.752-779, November, 1996.

This part covers the B-Prolog programming environment and all the built-ins available in B-Prolog. Considerable efforts have been made to make B-Prolog compliant with the standard. All possible discrepancies are explicitly described in this manual. In addition to the built-ins in the standard, B-Prolog also supports the built-ins in Dec-10 Prolog and some new ones such as those on arrays and hashtables.

This part is kept as compact as possible. The reader is referred to The Prolog Standard[2] for details about the built-ins in the standard, and to textbooks[3] or Web pages[4] for the basics of logic programming.

Part-II Agent and Constraint Programming

Prolog adopts a static computation rule that selects subgoals strictly from left to right. No subgoals can be delayed and no subgoals can be responsive to events. Prolog-II provides a predicate called `freeze`. The subgoal `freeze(X,p(X))` is logically equivalent to `p(X)` but the execution of `p(X)` is delayed until X is instantiated. B-Prolog provides a more powerful construct, called action rules, for programming agents. An agent is a subgoal that can be delayed and can be later activated by an event. Each time an agent is activated, some actions may be executed. Agents are a more general notion than freeze in Prolog-II and processes in concurrent logic programming in the sense that agents can be responsive to various kinds of events including user-defined ones.

Constraints are relationships among variables over some domain. B-Prolog supports constraints over trees, finite-domains, Boolean, and integer sets. In B-Prolog, constraint propagation is used to solve constraints. Each constraint is compiled into one or more agents, called constraint propagators, that are responsible for maintaining the consistency of the constraint. A constraint propagator is activated when the domain of any variable in the constraint is updated.

Action rules comprise a powerful and efficient language for programming constraint propagators, concurrent agents, event handlers, and interactive user interfaces. Action rules are unique to B-Prolog and are thus described in detail in the manual.

Part-III Graphics Programming

The widespread use of window systems has made a graphics package indispensable for any programming languages. Prolog and CLP languages in general are not an exception. It is possible to use the graphics packages in C++ or Java through the external language interfaces. This approach, however, is not satisfactory for the following two reasons. First, the programmer has to write code in two languages. This is especially daunting when there are interactions involved. Second, the graphics packages are at such a low level that they do not match well with a

[2] http://www.logic-programming.org/prolog_std.html

[3] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1994, Ivan Bratko, *Prolog for Artificial Intelligence*, Addison-Wesley, 2000, and
L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, 1997.

[4] e.g., http://archive.comlab.ox.ac.uk/logic-prog.html and http://www.cs.unh.edu/ccc/archive/.

high-level language like CLP.

The CGLIB in B-Prolog is implemented in B-Prolog, Java, and C. CGLIB enables the user to use the Java graphics package without the need to write any code in Java. CGLIB, however, is not just another syntax sugar for Java's graphics package. It has a significantly higher abstraction level and is much easier to learn and use than Java's graphics package.

CGLIB can be used in many areas such as drawing editors, interactive user interfaces, document authoring, animation, information visualization, intelligent agents, and games. This part, which is in a separate volume, also gives several sample applications in addition to the primitives in the library.

Acknowledgements

Since the software was released in March 1994, it has been greatly improved in performance, functionality, versatility, and portability. We appreciate the comments and contributions made by the users' group and the support by the CUNY Software Institute on related projects. Kenichi Katamine helped us port the system to Windows and notified us of Chris Thewalt's *getline* program which is used as the command reader in B-Prolog. Yi-Dong Shen inspired us to implement tabling in B-Prolog and shared the key idea with us. Many users ported B-Prolog to different platforms including HP-UX (Naoyuki Nide), AIX (Paul Eggert), FreeBSD (Shuhei Murakoshi), Linux, MacOS (Dietmar Planitzer and David Kopec), Epoc and Amiga(Tim Wentford). We encourage users' active involvement in this project. Please contact us if you are interested in porting B-Prolog to another platform or extending the functionality.

This distribution includes the following public domain modules: `read.pl` by D.H.D. Warren and Richard O'Keefe; `token.c`, `setof.pl` and `dcg.pl` by Richard O'Keefe; and `getline.c` by Chris Thewalt. The loader, `loader.c`, is based on the one used in SB-Prolog which was developed by the Logic Programming Group at SUNY Stony Brook. The Java interface is based on JIPL developed by Nobukuni Kino. The copyrights for these programs are owned by their authors as written in the sources.

Contents

Chapter 1

Getting Started with B-Prolog

1.1 How to enter and quit B-Prolog

Just like most Prolog systems, B-Prolog offers the user an interactive programming environment for compiling, loading, debugging and running programs. To enter the system, type the command:

```
bp
```

To start the system with the graphics package, use the command:

```
bpp
```

After the system is started, it responds with the prompt `|?-` and is ready to accept Prolog queries. The command `help` shows part of the commands that the system accepts.

```
help
```

To quit the system, use the query:

```
halt
```

or simply enter `^D` (control-D) when the cursor is located at the beginning of an empty line.

1.2 The command line editor

The command line editor resides at the top-level of the system accepting queries from the user. A query is a Prolog goal ended with a new line. It is a tradition that a period is used to terminate a query. In B-Prolog, as no query can expand over more than one line, the terminating period can be omitted.

The command line editor accepts the following editing commands:

`^F`	Move the cursor one position forward.
`^B`	Move the cursor one position backward.
`^A`	Move the cursor to the beginning of the line.
`^E`	Move the cursor to the end of the line.
`^D`	Delete the character under the cursor.
`^H`	Delete the character to the left of the cursor.
`^K`	Delete the characters to the right of the cursor.
`^U`	Delete the whole line.
`^P`	Load the previous query in the buffer.
`^N`	Load the next query in the buffer.

Notice that as mentioned above the command `^D` will halt the system if the line is empty and the cursor is located in the beginning of the line.

1.3 How to run programs

A program consists of a set of predicates. A predicate is made up of a sequence (not necessarily consecutive) of clauses whose heads have the same predicate symbol and the same arity. Each predicate is defined in one module stored in a file unless it is declared to be *dynamic*.

Compiling and loading

A program needs to be first compiled before being loaded into the system for execution. To compile a program in a file named `file-name`, type

```
compile(file-name).
```

If the file name has the extension `pl`, then the extension can be omitted. The compiled byte-code will be stored in a new file with the same primary name and the extension `out`. To load a compiled byte-code program, type

```
load(file-name).
```

Consulting

Another way to run a program is to consult it directly into the program area without compilation. It is possible to trace the execution of consulted programs but not compiled ones. To consult a program in a file into the program area, type

```
consult(file-name)
```

or simply

```
[file-name].
```

To see the consulted or dynamically asserted clauses in the program area, use

```
listing
```

and to see the clauses defining a predicate `Atom/Arity`, use

```
listing(Atom/Arity)
```

Running programs

After a program is loaded, the user can query the program. For each query, the system executes the program and reports `yes` when the query succeeds or `no` when the query fails. When a query that contains variables succeeds, the system also reports the bindings for the variables. The user can ask the system to find the next solution by typing `';'` after a solution.

Example:

```
?- member(X,[1,2,3]).
X=1;
X=2;
X=3;
no
```

The call `abort` stops the current execution and restores the system to the top-level.

Chapter 2

Programs

This chapter describes the syntax of Prolog. Both programs and data are composed from *terms* in Prolog.

2.1 Terms

A term is either a *constant*, a *variable*, or a *compound* term. There are two kinds of constants: *atoms* and *numbers.*

Atoms

Atoms are strings of letters, digits, and underscore marks _ that begin with a lower-case letter, or strings of any characters enclosed in single quotation marks. No atom can stretch over one line in a program and no atom can contain more than 1000 characters. The single quotation mark is also used as the escape character. So, the atom $'a''b'$ contains three characters, namely `a`, `'`, and `b`.

Numbers

A number is either an integer or a floating-point number. A decimal integer is a sequence of decimal digits with an optional sign preceding it. The range of integers is from $-2^{27}+1 \ = \ -268435455$ to $2^{27}-1 \ = \ 268435455$, inclusive.

An integer can be in the radix notation with a base other than 10. In general, an integer in the radix notation takes the form <`base`>'<`digits`> where `base` is a decimal integer and `digits` is a sequence of digits. If the `base` is zero, then the notation represents the code of the character following the single quotation mark.

Examples:

- `2'100` : 4 in binary notation.
- `8'73` : 59 in octal notation.

- `16'f7`: 247 in hexadecimal notation.
- `0'a`: the code of `'a'`, which is 97.

A floating-point number consists of an integer (optional), then a decimal point and then another integer[1]. For example, `23.2` and `0.23` are valid floating-point numbers.

Variables

Variables look like atoms, except they have names beginning with a capital letter or an underscore mark. A single underscore mark denotes an anonymous variable.

Compound terms

A compound term is a structure that takes the form of $f(t_1, \ldots, t_n)$ where n is called the arity, and f called the functor, or function symbol, and $t_1, \ldots, t_n$ are terms. In B-Prolog, the arity must be greater than 0 and less than 32768. The terms enclosed in the parentheses are called *components* of the compound term.

Lists are special structures whose functors are `'.'`. The special atom `'[]'` denotes an empty list. The list `[H|T]` denotes the structure '.'(H,T).

A string is represented as a list of codes of the characters in the string. For example, the string `"abc"` is the same as the list `[97,98,99]`. The double quotation mark is also used as the escape character for strings. So, the string `"a""c"` is the same as `[97,34,98]` where `34` is the code for the double quotation mark.

Arrays and hashtables are also represented as structures. All built-ins on structures can be also applied to arrays and hashtables. It is suggested, however, that only primitives on arrays and hashtables be used to manipulate them.

2.2 Programs

A program is a sequence of logical statements, called *Horn clauses*, of three types: *facts*, *rules*, and *directives*.

Facts

A fact is an *atomic formula* of the form $p(t_1, t_2, \ldots, t_n)$ where p is an n-ary predicate symbol and $t_1, t_2, \ldots, t_n$ are terms which are called the *arguments* of the atomic formula.

[1]Exponential notation of floating-point numbers is not supported currently.

Rules

A rule takes the form of

```
H :- B1,B2,...,Bn.  (n>0)
```

where `H, B1, ..., Bn` are atomic formulas. `H` is called the *head* and the right hand side of `:-` is called the *body* of the rule. A fact can be considered a special kind of rule whose body is `true`.

A *predicate* is an ordered sequence of clauses whose heads have the same predicate symbol and the same arity.

Directives

A directive gives a query that is to be executed when the program is loaded or tells the system some pragmatic information about the predicates in the program. A directive takes the form of

```
:- B1,B2,...,Bn.
```

where `B1, ..., Bn` are atomic formulas.

2.3 Control constructs

In Prolog, backtracking is employed to explore the search space for a query against a program. Goals in the query are executed from left to right, and the clauses in each predicate are tried sequentially from the top. A query may succeed, may fail, or may be terminated because of exceptions. When a query succeeds, the variables in it may be bound to some terms. The call `true` always succeeds, and the call `fail` always fails. There are several control constructs for controlling backtracking, for specifying *conjunction*, *negation*, *disjunction*, and *if-then-else*, and for finding *all solutions*.

Cut

Prolog provides an operator, called *cut*, for controlling backtracking. A cut is written as ! in programs. A cut in the body of a clause has the effect of removing the choice points, or alternative clauses, of the goals to the left of it.

Example:

The query `p(X)` against the following program only gives one solution `p(1)`. The cut removes the choice points for `p(X)` and `q(X)`, and thus no further solution will be returned when the user forces backtracking by typing ';'. Without the cut, the query `p(X)` would have three solutions.

```
p(X):-q(X),!.
p(3).

q(1).
q(2).
```

When a failure occurs, the execution will backtrack to the latest choice point, i.e., the latest subgoal that has alternative clauses. There are two non-standard built-ins, called `savecp/1` and `cutto/1`, which can make the system backtrack to a choice point deep in the search tree. The call `savecp(Cp)` binds `Cp` to the latest choice point frame, where `Cp` must be a variable. The call `cutto(Cp)` discards all the choice points created after `Cp`. In other words, the call lets `Cp` be the latest choice point. Notice that `Cp` must be a reference to a choice point set by `savecp(Cp)`.

Conjunction, disjunction, negation, and if-then-else

The construct `(P,Q)` denotes conjunction. It succeeds if both `P` and `Q` succeed.

The construct `(not P)` and `\+ P` denote negation. It succeeds if and only if `P` fails. No negation is transparent to cuts. In other words, the cuts in a negation are effective only in the negation. No cut in a negation can remove choice points created for the goals to the left of the negation.

The construct `(P;Q)` denotes disjunction. It succeeds if either `P` or `Q` succeeds. Q is executed only after `P` fails. Disjunction is transparent to cuts. A cut in `P` or `Q` will remove not only the choice points created for the goals to the left of the cut in `P` or `Q` but also the choice points created for the goals to the left of the disjunction.

The control construct `(If->Then;Else)` succeeds if (1) `If` and `Then` succeed, or (2) `If` fails and `Else` succeeds. `If` is not transparent to cuts, but `Then` and `Else` are transparent to cuts. The control construct `(If->Then)` is equivalent to `(If->Then;fail)`.

repeat/0

The predicate `repeat`, which is defined as follows, is a built-in predicate that is often used to express iterations.

```
repeat.
repeat:-repeat.
```

For example, the query

```
repeat,write(a),fail
```

repeatedly outputs 'a's until the user types control-c to stop it.

call/1 and once/1

The `call(Goal)` treats `Goal` as a subgoal. It is equivalent to `Goal`. The call `once(Goal)` is equivalent to `Goal` but can only succeed at most once. It is implemented as follows:

```
once(Goal):-call(Goal),!.
```

All solutions

- `findall(Term,Goal,List)` Succeeds if `List` is the list of instances of `Term` such that `Goal` succeeds. Example:

  ```
  ?-findall(X,member(X,[(1,a),(2,b),(3,c)]),Xs)
    Xs=[(1,a),(2,b),(3,c)]
  ```

- `bagof(Term,Goal,List)` The same as `findall(Term,Goal,List)` except for its treatment of free variables that do not occur in `Term` but in `Goal`. It will first pick the first tuple of values for the free variables and then use this tuple of values to find the list of solutions `List` of `Goal`. Example:

  ```
  ?-bagof(Y,member((X,Y),[(1,a),(2,b),(3,c)]),Xs)
    X=1
    Y=[a];
    X=2
    Y=[b];
    X=3
    Y=[c];
    no
  ```

- `setof(Term,Goal,List)` Like `bagof(Term,Goal,List)` but the elements of *List* are sorted into alphabetical order.

Chapter 3

Data Types and Built-ins

A data type is a set of values and a set of predicates on the values. The following depicts the containing relationship of the types available in B-Prolog.

- term
  - atom
  - number
    - integer
    - floating-point number
  - variable
  - compound term
    - structure
    - list
    - array
    - hashtable

The B-Prolog system provides a set of built-in predicates for each of the types. Built-ins cannot be redefined.

3.1 Terms

The built-ins described in this section can be applied to any type of terms.

3.1.1 Type checking

- `atom(X)` The term `X` is an atom.
- `atomic(X)` The term `X` is an atom or a number.
- `float(X)` The term `X` is a floating-point number.
- `real(X)` The same as `float(X)`.

- `integer(X)` The term `X` is an integer.
- `number(X)` The term `X` is a number.
- `nonvar(X)` The term `X` is not a variable.
- `var(X)` The term `X` is a free variable.
- `compound(X)` The term `X` is a compound term. It is true if `X` is either a structure or a list.
- `ground(X)` The term `X` is ground.

3.1.2 Unification

- `X = Y` The terms `X` and `Y` are unified.
- `X \= Y` The terms `X` and `Y` are not unifiable.
- `X?=Y` The terms `X` and `Y` are unifiable. It is logically equivalent to: `not(not(X=Y))`.

3.1.3 Term comparison and manipulation

- `Term1 == Term2` The terms `Term1` and `Term2` are strictly identical.
- `Term1 \== Term2` The terms `Term1` and `Term2` are not strictly identical.
- `Term1 @=< Term2` The term `Term1` precedes or is identical to the term `Term2` in the standard order.
- `Term1 @> Term2` The term `Term1` follows the term `Term2` in the standard order.
- `Term1 @>= Term2` The term `Term1` follows or is identical to the term `Term2` in the standard order.
- `Term1 @< Term2` The term `Term1` precedes the term `Term2` in the standard order.
- `compare(Op,Term1,Term2)` `Op` is the result of comparing the terms `Term1` and `Term2`.
- `copy_term(Term,CopyOfTerm)` `CopyOfTerm` is an independent copy of `Term`.
- `number_vars(Term,N0,N)`
- `numbervars(Term,N0,N)` Number the variables in `Term` by using the integers starting from `N0`. `N` is the next integer available after the term is numbered. Let `N0`, `N1`, ..., `N-1` be the sequence of integers. The first variable is bound to the term `$var(N0)`, the second is bound to `$var(N1)`, and so on. Different variables receive different numberings and the occurrences of the same variable all receive the same numbering. *(not in ISO).*

- `unnumber_vars(Term1,Term2)` `Term2` is a copy of `Term1` with all numbered variables `$var(N)` being replaced by Prolog variables. Different numbered variables are replaced by different Prolog variables.

  Number the variables in `Term` by using the integers starting from `N0`. `N` is the next integer available after the term is numbered. Let `N0`, `N1`, ..., `N-1` be the sequence of integers. The first variable is bound to the term `$var(N0)`, the second is bound to `$var(N1)`, and so on. Different variables receive different numberings and the occurrences of the same variable all receive the same numbering. *(not in ISO)*.

- `vars_set(Term,Set)` `Set` is a list of variables that occur in `Term`.

3.2 Numbers

An arithmetic expression is a term built from numbers, variables, and the arithmetic functions listed in Table 3.1. An expression must be ground when it is evaluated.

- `Exp1 is Exp2` The term `Exp2` must be a ground expression and `Exp1` must be either a variable or a ground expression. If `Exp1` is a variable, then the call binds the variable to the result of `Exp2`. If `Exp1` is a non-variable expression, then the call is equivalent to `Exp1 =:= Exp2`.
- `X =:= Y` The expression `X` is numerically equal to `Y`.
- `X =\= Y` The expression `X` is not numerically equal to `Y`.
- `X < Y` The expression `X` is less than `Y`.
- `X =< Y` The expression `X` is less than or equal to `Y`.
- `X > Y` The expression `X` is greater than `Y`.
- `X >= Y` The expression `X` is greater than or equal to `Y`.

3.3 Lists and structures

- `Term =.. List` The functor and arguments of `Term` comprise the list `List`.
- `append(L1,L2,L3)` True when `L3` is the concatenation of `L1` and `L2`. *(not in ISO)*.
- `arg(ArgNo,Term,Arg)` The `ArgNo`th argument of the term `Term` is `Arg`.
- `functor(Term,Name,Arity)` The principal functor of the term `Term` has the name `Name` and arity `Arity`.

`X + Y`	addition.
`X - Y`	subtraction.
`X * Y`	multiplication.
`X / Y`	division.
`X // Y`	integer division.
`X mod Y`	modulo.
`rem(X,Y)`	remainder (X-(X//Y)*Y).
`X /> Y`	integer division (ceiling(X/Y)).
`X /< Y`	integer division (floor(X/Y)).
`X ** Y`	power.
`-X`	sign reversal.
`X >> Y`	bit shift right.
`X << Y`	bit shift left.
`X /\ Y`	bit wise and.
`X \/ Y`	bit wise or.
`\ X`	bit wise complement.
`abs(X)`	absolution value.
`atan(X)`	arctangent(argument in radians).
`ceiling(X)`	smallest integer not smaller than `X`.
`cos(X)`	cosine (argument is radians).
`exp(X)`	natural antilogarithm, e^X.
`integer(X)`	convert `X` to integer.
`float(X)`	convert `X` to float.
`floor(X)`	largest integer not greater than `X`.
`log(X)`	natural logarithm, $log_e X$.
`max(X,Y)`	the maximum of `X` and `Y` *(not in ISO).*
`min(X,Y)`	the minimum of `X` and `Y` *(not in ISO).*
`pi`	the constant pi *(not in ISO).*
`random`	a random number *(not in ISO).*
`random(Seed)`	a random number generated by using `Seed` *(not in ISO).*
`round(X)`	integer nearest to `X`.
`sign(X)`	sign (-1 for negative, 0 for zero, and 1 for positive).
`sin(X)`	sine (argument in radians).
`sqrt(X)`	square root.
`truncate(X)`	integer part of `X`.

Figure 3.1: Arithmetic functions.

- `length(List,Length)` The length of list `List` is Length. *(not in ISO).*
- `membchk(X,L)` True when `X` is included in the list `L`. '==/2' is used to test whether two terms are the same. *(not in ISO).*
- `member(X,L)` True when `X` is a member of the list `L`. Instantiates `X` to different elements in `L` upon backtracking. *(not in ISO).*
- `reverse(L1,L2)` True when `L2` is the reverse of `L1`. *(not in ISO).*
- `setarg(ArgNo,CompoundTerm,NewArg)` Replaces destructively the `ArgNo`th argument of `CompoundTerm` with `NewArg`. The update is undone on backtracking. *(not in ISO).*
- `sort(List1,List2)` Sorting the list `List1` yields `List2`. *(not in ISO).*

3.4 Arrays *(not in ISO)*

An array is a collection of elements. As array elements can be arrays, arrays can be multi-dimensional. Arrays are created by the built-in predicate `new_array(X,Ranges)` where `X` must be an uninstantiated variable and `Ranges` must be a list of positive integers. Let `Ranges` be `[N1,...,Nn]`. Then, `N1` is the size of the first dimension, `N2` is the size of the second dimension, and so on. For example, the call `new_array(X,[10,20])` binds `X` to a two dimensional array where the first dimension has 10 elements and the second dimension has 20 elements. The 200 array elements are free variables when the array is created.

The operator `@=` is provided for initializing and accessing arrays. The call `A^[I] @= Elm` unifies the `I`th element of `A` with `Elm`. The index `I` must be an expression whose value is from 1 to the size of the array. A type exception will be raised if `A` is not an array. This notation extends to multi-dimensional arrays. In general, the call `A^[I1,...,In] @= Elm` is equivalent to:

```
A^[I1] @= T, T^[I2,...,In] @= Elm
```

The operator `@=` can also be used to initialize arrays. For example,

```
new_array(X,[3]), X @= [1,2,3]
```

creates a one-dimensional array with three integers 1, 2, and 3, and the following

```
new_array(X,[2,2]), X @= [[1,2],[3,4]]
```

creates a two-dimensional array with four integers. Notice that lists or lists of lists are not treated as arrays in other contexts.

- `new_array(X,Ranges)` X is an array whose dimension sizes are specified by `Ranges`.
- `X^length @= Length` The length of the array `X` is `Length`. If `X` is a multi-dimensional array, then `Length` is the size of the first dimension.

- `X^dimension @= Dim` `Dim` is the dimension of `X`.
- `X^rows @= Rows` `Rows` is a list of rows in the array `X`. The dimension of `X` must be no less than 2.
- `X^columns @= Cols` `Cols` is a list of columns in the array `X`. The dimension of `X` must be no less than 2.
- `is_array(A)` Succeeds if `A` is an array.
- `X^Indexes @= Elm` The element at `Indexes` of the array `X` is `Elm`. `Indexes` must be a list of integers `[I1,I2,...,In]` where `Ii` must be an integer in the range from `1` to the size of the corresponding dimension. Notice that indexes start from 1, which is different from in many other languages.
- `X^Indexes @:= Elm` Destructively replace the element at `Indexes` of the array `X` by `Elm`. The update is undone upon backtracking.
- `array_to_list(X,List)` The term `List` is a list of all elements in array `X`. Suppose `X` is an `n` dimensional array and the sizes of the dimensions are `N1`, `N2`, ..., and `Nn`. Then `List` contains the elements from `[1,...,1]`, `[1,...,2]`, to `[N1,N2,...,Nn]`.

3.5 Hashtables *(not in ISO)*

- `new_hashtable(T)` Create a hashtable `T` with `7` bucket slots.
- `new_hashtable(T,N)` Create a hashtable `T` with `N` bucket slots. `N` must be a positive integer.
- `hashtable_put(T,Key,Value)` Put `Value` under the key `Key` to hashtable `T`.
- `hashtable_get(T,Key,Value)` Get `Value` that has the key `Key` from hashtable `T`. Fail if no such a value exists.
- `hashtable_size(T,Size)` The size of hashtable `T`, i.e., the number of bucket slots, is `Size`.
- `hash_code(Term,Code)` The hash code of `Term` is `Code`.
- `hashtable_to_list(T,List)` `List` is the list of key and value pairs in hashtable `T`.
- `hashtable_keys_to_list(T,List)` `List` is the list of keys of the elements in hashtable `T`.
- `hashtable_values_to_list(T,List)` `List` is the list of values of the elements in hashtable `T`.

3.6 Character-string operations

- `atom_chars(Atom,Chars)` `Chars` is the list of characters of `Atom`.
- `atom_codes(Atom,Codes)` `Codes` is the list of numeric codes of the characters of `Atom`.
- `atom_concat(Atom1,Atom2,Atom3)` The concatenation of `Atom1` and `Atom2` is equal to `Atom3`. Either both `Atom1` and `Atom2` are atoms or `Atom3` is an atom.
- `atom_length(Atom,Length)` Length (in characters) of `Atom` is `Length`.
- `char_code(Char,Code)` The numeric code of the character `Char` is `Code`.
- `number_chars(Num,Chars)` `Chars` is the list of digits (including '.') of the number `Num`.
- `number_codes(Num,Codes)` `Codes` is the list of numeric codes of digits of the number `Num`.
- `sub_atom(Atom,PreLen,Len,PostLen,Sub)` The atom `Atom` is divided into three parts, `Pre`, `Sub`, and `Post` with respective lengths of `PreLen`, `Len`, and `PostLen`.
- `name(Const,CharList)` The name of atom or number `Const` is the string `CharList`. *(not in ISO)*.
- `parse_atom(Atom,Term,Vars)` Convert `Atom` to `Term` where `Vars` is a list of elements in the form `(VarName=Var)`. It fails if `Atom` is not syntactically correct. Examples:

  ```
  | ?- parse_atom('X is 1+1',Term,Vars)
  Vars = [X=_8c019c]
  Term = _8c019c is 1+1?

  | ?- parse_atom('p(X,Y),q(Y,Z)',Term,Vars)
  Vars = [Z=_8c01d8,Y=_8c01d4,X=_8c01d0]
  Term = p(_8c01d0,_8c01d4),q(_8c01d4,_8c01d8)?

  | ?- parse_atom(' a b c',Term,Vars)
  *** syntax error ***
  a <<here>> b c
  no
  ```

  (not in ISO).
- `parse_atom(Atom,Term)` Equivalent to `parse_atom(Atom,Term,_)`

- `parse_string(String,Term,Vars)` Similar to `parse_atom` but the first argument is a list of codes. Example:

  ```
  | ?- name('X is 1+1',String),parse_string(String,Term,Vars)
  Vars = [X=_8c0294]
  Term = _8c0294 is 1+1
  String = [88,32,105,115,32,49,43,49]?
  ```

  (not in ISO).

- `parse_string(String,Term)` Equivalent to `parse_string(String,Term,_)`.

- `term2atom(Term,Atom)` `Atom` is an atom that encodes `Term`. Example:

  ```
  | ?- term2atom(f(X,Y,X),S),writeq(S),nl.
  'f(_9250158,_9250188,_9250158)'
  S=f(_9250158,_9250188,_9250158)
  ```

  (not in ISO).

- `term2string(Term,String)` Equivalent to:

  ```
  term2atom(Term,Atom),atom_codes(Atom,String)
  ```

  (not in ISO).

- `write_string(String)` Write the list of codes `String` as a readable string. For example, `write_string([97,98,99])` outputs `"abc"`. *(not in ISO).*

Chapter 4

Exception Handling

4.1 Exceptions

In addition to success and failure, a program may give an exception that is thrown explicitly by a call of `throw/2` or raised by a built-in or caused by the user's typing of control-c. An exception raised by a built-in is an one-argument structure where the functor tells the type and the argument tells the source of the exception. The following lists some of the exceptions:

- `divide_by_zero(Goal)`: `Goal` divides a number by zero.
- `file_not_found(Goal)`: `Goal` tries to open a file that does not exist.
- `illegal_arguments(Goal)`: `Goal` has an illegal argument.
- `number_expected(Goal)`: `Goal` evaluates an invalid expression.
- `out_of_range(Goal)`: `Goal` tries to access an element of a structure or an array using an index that is out of range.

The exception caused by the typing of control-c is an atom named `interrupt`.

An exception that is not caught by the user's program will be handled by the system. The system reports the type and the source of the exception, displays the chain of calls that led to the exception, and aborts execution of the query. For example, for the query `a=:=1`, the system will report:

```
** Error ** number_expected: a=:=1
```

where `number_expected` is the type and `a=:=1` is the source.

4.2 throw/1

A user's program can throw exceptions too. The call `throw(E)` raises an exception `E` to be caught and handled by some ancestor catcher or handler. If there is no catcher available in the chain of ancestor calls, the system will handle it.

4.3 `catch/3`

All exceptions including those raised by built-ins and interruptions can be caught by catchers. A catcher is a call in the form:

```
catch(Goal,ExceptionPattern,Recovergoal)
```

which is equivalent to `Goal` except when an exception is raised during the execution of `Goal` that unifies `ExceptionPattern`. When such an exception is raised, all the bindings that have been performed on variables in `Goal` will be undone and `Recovergoal` will be executed to handle the exception. Notice that `ExceptionPattern` is unified with a renamed copy of the exception before `Recovergoal` is executed. Notice also that only exceptions that are raised by a descendant call of `Goal` can be caught.

Examples:

- `q(X)`, which is defined in the following, is equivalent to `p(X)` but all interruptions are ignored.

```
q(X):-catch(p(X),interrupt,q(X)).
```

- The query `catch(p(X),undefined_predicate(_),fail)` fails `p(X)` if an undefined predicate is called during its execution.
- The query `catch(q,C,write(hello_q))`, where `q` is defined in the following, succeeds with the unifier `C=c` and the message `hello_q`.

```
q :- r(c).
r(X) :- throw(X).
```

- The query `catch(p(X),E,p(X)==E)` against the following program fails because `E` is unified with a renamed copy of `p(X)` rather than `p(X)` itself.

```
p(X):-throw(p(X)).
```

Chapter 5

Directives and Prolog Flags

Directives inform the compiler or interpreter of some information about the predicates in a program[1].

5.1 Mode declaration

For Edinburgh style programs, the programmer can provide the compiler with modes to help it generate efficient code[2]. The mode of a predicate p indicates how the arguments of any call to p are instantiated just before the call is evaluated. The mode of a predicate p of n arguments is declared as

:-mode p($M1$,...,Mn).

where Mi is c (or $+$), f (or $-$), nv, d (or ?), or a structured mode. The mode c means a closed term that cannot be changed by the predicate; f means a free variable; nv means a non-variable term; and d means a don't-know term. The structured mode $l(M1, M2)$ means a list whose head and tail have modes $M1$ and $M2$ respectively; the structured mode $s(M1, \ldots Mn)$ means a compound term whose arguments have modes $M1$, ..., and Mn respectively.

5.2 include/1

The directive

```
:-include(File).
```

[1]The directives `multifile/1`, `discontiguous/1`, and `char_conversion/2` in ISO-Prolog are not supported currently. If a predicate is defined in multiple files or is discontiguous, it must be declared dynamic; otherwise, only part of the definition is effective. A clause in the form `:-Goal`, where `Goal` is none of the directives described here, specifies a query to be executed after the program is loaded or consulted. For example, the clause `:-op(Priority,Specifier,Atom)` will invoke the built-in predicate `op/3` and change the atom `Atom` into an operator with properties as specified by `Specifier` and `Priority`.

[2]In version 6.0, mode declarations are ignored by the compiler since wrong mode declarations can cause vague bugs.

will be replaced by the directives and clauses in `File` which must be a valid Prolog text file. The extension name can be omitted if it is `pl`.

5.3 Initialization

The directive

```
:-initialization(Goal).
```

is equivalent to:

```
:-Goal.
```

unless `Goal` is a directive. It specifies that as soon as the program is loaded or consulted, the goal `Goal` is to be executed.

5.4 Dynamic declaration

A predicate is either static or dynamic. Static predicates cannot be updated during execution. Dynamic predicates are stored in consulted form, and can be updated during execution. Predicates are assumed to be static unless they are explicitly declared to be dynamic. To declare predicates to be dynamic, use the following declaration:

```
:-dynamic Atom/Arity,...,Atom/Arity.
```

5.5 Table declaration

A tabled predicate is a predicate for which answers will be memorized in a table and variant calls of the predicate will be resolved by using the answers. The declaration,

```
:-table P1/N1, ..., Pk/Nk.
```

declares that the predicates `Pi/Ni` (i=1,...,k) are tabled predicates.

5.6 Prolog flags

A flag is an atom with an associated value. There are five flags being supported currently:

- `unknown`The value is either `fail`, meaning that calls to undefined predicates will be treated as failure, or `error`, meaning that an exception will be raised. The default value for the flag is `error`.

- `singleton` This flag governs whether warning messages about singleton variables will be emitted or not. The value is either `on` or `off`, and the default value is `on`.
- `redefined` This flag governs whether warning messages about redefined predicates are emitted or not. The value is either `on` or `off`, and the default value is `on`.
- `gc` Turn `on` or `off` the garbage collector (see Garbage collection).
- `gc_threshold` Set a new threshold constant (see Garbage collection).
- `action_warning` Issue a warning message when the action of an action rule fails. The default value is `on`.

The user can change the value of a flag to affect the behavior of the system and access the current value of a flag.

- `set_prolog_flag(Flag,Value)` Set value of `Flag` to be `Value`.
- `current_prolog_flag(Flag,Value)` `Value` is the current value of `Flag`.

Chapter 6

Debugging

6.1 Execution modes

There are two execution modes: *usual mode* and *debugging mode*. The query

```
trace
```

switches the execution mode to the debugging mode, and the query

```
notrace
```

switches the execution mode back to the usual mode. In debugging mode, the execution of asserted and consulted clauses can be traced. To trace part of the execution of a program, use `spy` to set spy points.

```
spy(Atom/Arity).
```

The spy points can be removed by

```
nospy
```

To remove only one spy point, use

```
nospy(Atom/Arity)
```

6.2 Debugging commands

In debugging mode, the system displays a message when a predicate is entered (Call), exited (Exit), reentered (Redo) or has failed (Fail). After a predicate is entered or reentered, the system waits for a command from the user. A command is a single letter followed by a carriage-return, or may simply be a carriage-return. The following commands are available:

- `RET` - This command causes the system to display a message at each step.
- `c`(reep) - the same as a carriage-return `RET`.

- `l`(eap) - causes the system to run in usual mode until a spy-point is reached.
- `s`(kip) - causes the system to run in usual mode until the predicate is finished (Exit or Fail).
- `r`(epeat creep) - causes the system to creep without asking for further commands from the user.
- `a`(bort) - causes the system to abort execution.
- `h`(elp) or `?` - causes the system to display available commands and their meaning.

Chapter 7

Input and Output

There are two groups of file manipulation predicates in B-Prolog. One group includes all input/output predicates described in the ISO draft for Prolog and the other group is inherited from DEC-10 Prolog. The latter is implemented by using the predicates in the former group.

7.1 Stream

A *stream* is a connection to a file. The user's terminal is treated as a special file. A stream can be referred to by a stream identifier or its aliases. By default, the streams `user_input` and `user_output` are already open, referring to the standard input (keyboard) and the standard output (screen) respectively.

- `open(FileName,Mode,Stream,Options)`
- `open(FileName,Mode,Stream)`
  Opens a file for input or output as indicated by I/O mode `Mode` and the list of stream-options `Options`. If it succeeds in opening the file, it unifies `Stream` with the stream identifier of the associated stream. If `FileName` is already opened, this predicate unifies `Stream` with the stream identifier already associated with the opened stream, but does not affect the contents of the file.

  An I/O mode is one of the following atoms:

  - `read` - Input. `FileName` must be the name of a file that already exists.
  - `write` - Output. If the file identified by `FileName` already exists, then the file is emptied; otherwise, a file with the name `FileName` is created.
  - `append` - Output. Similar to `write` except that the contents of a file will not be lost if it already exists.

  The list of stream-options is optional and can be empty or a list that includes[1]:

[1]The option `reposition(true)` in ISO-Prolog is not supported currently.

  - `type(text)` or `type(binary)`. The default is `type(text)`. This option does not have any effect on file manipulations.
  - `alias(Atom)`. Gives the stream the name `Atom`. A stream-alias can appear anywhere a stream can occur. A stream can be given multiple names, but an atom cannot be used as the name of more than one stream.
  - `eof_action(Atom)`. Specifies what to do upon repeated attempts to read past the end of the file. `Atom` can be[2]:
    - `error` - raises an error condition.
    - `eof_code` (the default)- makes each attempt return the same code that the first one did (`-1` or `end_of_file`).

- `close(Stream,Options)`
- `close(Stream)`
  Closes a stream identified by `Stream`, a stream identifier or a stream alias. The `Options` can include:
  - `force(false)` - raises an error condition if an error occurs while closing the stream.
  - `force(true)` - succeeds in any case.
- `stream_property(Stream,Property)` It is true if the stream identified by the stream identifier or stream alias `Stream` has a stream property `Property`. `Property` may be one of the following[3]:
  - `file_name(Name)` - the file name.
  - `mode(M)` - `input` or `output`.
  - `alias(A)` - `A` is the stream's alias if any.
  - `end_of_stream(E)` - where `E` is `at`, `past` or `no`, indicating whether reading has just reached the end of file, has gone past it or has not reached it.
  - `eof_action(A)` - action taken upon reading past the end of file.
  - `type(T)` - `T` is the type of the file.
- `current_input(Stream)` It is true if the stream identifier or stream alias `Stream` identifies the current input stream.
- `current_output(Stream)` It is true if the stream identifier or stream alias `Stream` identifies the current output stream.
- `set_input(Stream)` Sets the stream identified by `Stream` to be the current input stream.

[2]the option `eof_action(reset)` in ISO-Prolog is not supported currently.
[3]`position(P)` and `reposition(B)` in ISO-Prolog are not supported currently.

- `set_output(Stream)` Sets the stream identified by `Stream` to be the current output stream.
- `flush_output` Sends any output which is buffered for the current output stream to that stream.
- `flush_output(Stream)` Sends any output which is buffered for the stream identified by `Stream` to the stream.
- `at_end_of_stream` It is true if reading the current input stream has reached the end of file or is past the end of file.
- `at_end_of_stream(Stream)` It is true if reading the input stream `Stream` has reached the end of file or is past the end of file.

7.2 Character input/output

- `get_char(Stream,Char)` Inputs a character (if `Stream` is a text stream) or a byte (if `Stream` is a binary stream) from the stream `Stream` and unifies it with `Char`. After reaching the end of file, it unifies `Char` with `end_of_file`.
- `get_char(Char)` The same as the previous one except that the current input stream is used.
- `peek_char(Stream,Char)` The current character in `Stream` is `Char`. The position pointer of `Stream` remains the same after this operation.
- `peek_char(Char)` The same as `peek_char(Stream,Char)` except that the current input stream is used.
- `put_char(Stream,Char)` Outputs the character `Char` to the stream `Stream`.
- `put_char(Char)` Outputs the character `Char` to the current output stream.
- `nl(Stream)` Outputs the new line character to the stream `Stream`.
- `nl` Outputs the new line character to the current output stream.

7.3 Character code input/output

- `get_code(Stream,Code)` Inputs a byte from `Stream` and unifies `Code` with the byte. After reaching the end of file, it unifies `Code` with `-1`.
- `get_code(Code)` The same as the previous one except that the current input stream is used.
- `peek_code(Stream,Code)` The current code in `Stream` is `Code`. The postion pointer of `Stream` remains the same after this operation.

- peek_code(Code) The same as
- peek_code(Stream,Code) except that the current input stream is used.
- put_code(Stream,Code) Outputs a byte Code to the stream Stream.
- put_code(Code) Outputs a byte Code to the current output stream.

7.4 Byte input/output

- get_byte(Stream,Byte) Inputs a byte from Stream and unifies Byte with the byte. After reaching the end of file, it unifies Byte with -1.
- get_byte(Byte) The same as the previous one except that the current input stream is used.
- peek_byte(Stream,Byte) The current byte in Stream is Byte. The postion pointer of Stream remains the same after this operation.
- peek_byte(Byte) The same as
- peek_byte(Stream,Byte) except that the current input stream is used.
- put_byte(Stream,Byte) Outputs a byte Byte to the stream Stream.
- put_byte(Byte) Outputs a byte Byte to the current output stream.

7.5 Term input/output

These predicates[4] enable a Prolog term to be input from, or to be output to a stream. A term to be input must be followed by a period and then by white space.

- read_term(Stream,Term,Options) Inputs a term Term from the stream Stream using options Options. After reaching the end of file, it unifies Term with end_of_file. The Options is a list of options that can include:
  - variables(V_list) After reading a term, V_list will be unified with the list of variables that occur in the term.
  - variable_names(VN_list) After reading a term, VN_list will be unified with a list of elements in the form of N = V where V is a variable occurring in the term and N is the name of V.
  - singletons(VS_list) After reading a term, VS_list will be unified with a list of elements in the form N = V where V is a singleton variable in Term and N is its name.

[4]The predicates char_conversion/2 and current_char_conversion/2 in ISO-Prolog are not provided currently.

- `read_term(Term,Options)` The same as the previous one except that the current input stream is used.
- `read(Stream,Term)` Equivalent to: `read_term(Stream,Term),[])`.
- `read(Term)` Equivalent to: `read_term(Term,[])`.
- `write_term(Stream,Term,Options)` Outputs a term `Term` into a stream `Stream` using the option list `Options`. The list of options `Options` can include[5]:
  - `quoted(Bool)` - When `Bool` is `true` each atom and functor is quoted such that the term can be read by `read/1`.
  - `ignore_ops(Bool)` - When `Bool` is `true` each compound term is output in functional notation, i.e., in the form of `f(A1,...,An)` where `f` is the functor and `Ai` (i=1,...,n) are arguments.
- `write_term(Term,Options)` The same as the previous one except that the current output stream is used.
- `write(Stream,Term)` Equivalent to: `write_term(Stream,Term,[])`.
- `write(Term)` Equivalent to:

```
current_output(Stream),write(Stream,Term).
```

- `write_canonical(Stream,Term)` Equivalent to:

```
write_term(Stream,Term,[quoted(true),ignore_ops(true)]).
```

- `write_canonical(Term)` Equivalent to:

```
current_output(Stream),write_canonical(Stream,Term).
```

- `writeq(Stream,Term)` Equivalent to:

```
write_term(Stream,Term,[quoted(true)]).
```

- `writeq(Term)` Equivalent to:

```
current_output(Stream),writeq(Stream,Term).
```

- `op(Priority,Specifier,Name)` Makes atom `Name` an operator of type `Specifier` and priority `Priority`[6]. `Specifier` specifies the class (`prefix, infix or postfix`) and the associativity, which can be:

[5]The option `numbervars(Bool)` in ISO-Prolog is not supported currently.
[6]The predefined operator ',' can not be altered.

  - `fx` - prefix, non-associative.
  - `fy` - prefix, right-associative.
  - `xfx` - infix, non-associative.
  - `xfy` - infix, right-associative.
  - `yfx` - infix, left-associative.
  - `xf` - postfix, non-associative.
  - `yf` - postfix, left-associative.

  The priority of an operator is an integer greater than 0 and less than 1201. The lower the priority, the stronger the operator binds its operands.

- `current_op(Priority,Specifier,Operator)` It is true if `Operator` is an operator with properties defined by a specifier `Specifier` and precedence `Priority`.

7.6 Input/output of DEC-10 Prolog *(not in ISO)*

This section describes the built-in predicates for file manipulation inherited from DEC-10 Prolog. These predicates refer to streams by file names. The atom `user` is a reference to both the standard input and standard output streams.

- `see(FileName)` Makes the file `FileName` the current input stream. It is equivalent to:

  ```
  open(FileName,read,Stream),set_input(Stream).
  ```

- `seeing(File)` The current input stream is named `FileName`. It is equivalent to:

  ```
  current_input(Stream),stream_property(Stream,file_name(FileName)).
  ```

- `seen` Closes the current input stream. It is equivalent to:

  ```
  current_input(Stream),close(Stream).
  ```

- `tell(FileName)` Makes the file `FileName` the current output stream. It is equivalent to:

  ```
  open(FileName,write,Stream),set_output(Stream).
  ```

- `telling(FileName)` The current output stream is named `FileName`. It is equivalent to:

  ```
  current_output(Stream),
  stream_property(Stream,file_name(FileName).
  ```

- `told` Closes the current output stream. It is equivalent to:

```
current_output(Stream),close(Stream).
```

- `get(Code)` `Code` is the next printable byte code in the current input stream.
- `get0(Code)` `Code` is the next byte code in the current input stream.
- `put(Code)` Output the character to the current output stream, whose code is `Code`.
- `tab(N)` Outputs `N` spaces to the current output stream.
- `exists(F)` Succeeds if the file `F` exists.

7.7 Formatted output of terms

The predicate `format(Format,L)`, which mimics the `printf` function in C, prints the elements in the list `L` under the control of `Format`, a string of characters. There are two kinds of characters in `Format`: *normal* characters are output verbatim, and `control` characters formats the elements in `L`. Control characters all start with `~`. For example,

```
format("~thello~t world~t~a~t~4c~t~4d~t~7f",[atom,0'x,123,12.3])
```

give the following output:

```
hello    world  atom    xxxx     123        12.300000
```

The control characters `~a`, `~4c`,`~4d`, and `~7f` control the output of the atom `atom`, character `0'x`, integer `123`, and float `12.3`, respectively. The control characters `~t` put the data into different columns.

- `format(Format,L)`: Output the arguments in the list `L` under the control of `Format`.
- `format(Stream,Format,L)`: The same as `format(Format,L)` but it sends output to `Stream`.

The following control characters are supported:

- `~~`: Print `~`.
- `~N|`: Specifies a new position for the next argument.
- `~N+`: The same as `~N|`.
- `~a`: print the atom without quoting. Exception is raised if the argument is not an atom.

- `~Nc`: The argument must be a character code. Output the argument `N` times. Output the argument once if `N` is missing.
- `~Nf`,`~Ne`, `~Ng`: The argument must be a number. The C function `printf` is called to print the argument with the format `"%.Nf"`, `"%.Ne"`, and `"%.Ng"`, respectively. `".N''` does not occur in the format for the C function if `N` is not specified in the Prolog format.
- `~Nd`: The argument must be a number. `N` specifies the width of the argument. If the argument occupies more than `N` spaces, then enough spaces are filled to the left of the number.
- `~Ns`: The argument must be a list of character codes. Exactly N characters will be printed. Spaces are filled to the right of the string if the length of the string is less than `N`.
- `~k`: Pass the argument to write_canonical/1.
- `~p`: Pass the argument to print/1.
- `~q`: Pass the argument to writeq/1.
- `~w`: Pass the argument to write/1.
- `~Nn`: Print `N` new lines.
- `~t`: Move the position to the next column. Each column is assumed to be 8 characters long.

Chapter 8

Dynamic Clauses and Global Variables

This chapter describes predicates for manipulating dynamic clauses.

8.1 Predicates of ISO-Prolog

- `asserta(Clause)` Asserts `Clause` as the first clause in its predicate.
- `assertz(Clause)` Asserts `Clause` as the last clause in its predicate.
- `assert(Clause)` The same as `assertz(Clause)`
- `retract(Clause)` Removes from the predicate a clause that unifies `Clause`. Upon backtracking, removes the next unifiable clause.
- `abolish(Functor/Arity)` Completely removes the dynamic predicate identified by `Functor/Arity` from the program area.
- `clause(Head,Body)` It is true if `Head` and `Body` unify with the head and the body of a dynamically asserted (or consulted) clause. The body of a fact is `true`. Gives multiple solutions upon backtracking.
- `current_predicate(Functor/Arity)` It is true if `Functor/Arity` identifies a defined predicate, whether static or dynamic, in the program area. Gives multiple solutions upon backtracking.

8.2 Predicates of DEC-10 Prolog *(not in ISO)*

- `abolish` Removes all the dynamic predicates from the program area.
- `recorda(Key,Term,Ref)` Makes the term `Term` the first record under the key `Key` with a unique identifier `Ref`.

- `recorded(Key,Term,Ref)` The term `Term` is currently recorded under the key `Key` with a unique identifier `Ref`.
- `recordz(Key,Term,Ref)` Makes the term `Term` the last record under the key `Key` with a unique identifier `Ref`.
- `erase(Ref)` Erases the record whose unique identifier is `Ref`.

8.3 Global variables *(not in ISO)*

A global variable has a name `F/N` and a value associated with it. A name cannot be used at the same time as both a global variable name and a predicate name.

- `global_set(F,N,Value)` Set the value of the global variable `F/N` to `Value`. After this call, the name `F/N` becomes a global variable. If the name `F/N` was used as a predicate name, then all the information about the predicate will be erased.
- `global_set(F,Value)` Equivalent to `global_set(F,0,Value)`.
- `global_get(F,N,Value)` The value associated with the global variable `F/N` is `Value`. If `F/N` is not a global variable, then the call fails.
- `global_get(F,Value)` Equivalent to `global_get(F,0,Value)`.

Chapter 9

Memory Management and Garbage Collection

In the ATOAM, there are five data areas: *program area*, *heap*, *control stack*, *trail stack*, and *table area*. The *program area* contains, besides programs, a symbol table that stores information about the atoms, functions and predicate symbols in the programs. The *heap* stores terms created during execution. The *control* stack stores activation frames associated with predicate calls. The *trail* stack stores updates of those words that must be unbound upon backtracking. The *tail* area is used to store tabled subgoals and their answers.

9.1 Memory allocation

The shell file `bp` specifies the sizes (number of words) for the data areas. Initially, the following values are given:

```
set PAREA=2000000
set STACK=2000000
set TRAIL=1000000
set TABLE=20000
```

The `PAREA` is the size for the program area, `STACK` is the total size for the control stack and the heap, `TRAIL` is the size for the trail stack, and `TABLE` is the size for the table area. The user can freely update these values. The user can check the current memory consumption by using `statistics/0` or `statistics/2`.

The user can modify the shell script file to increase or decrease the amounts. The user can also specify the amount of space allocated to a stack when starting the system. For example,

```
bp -s 4000000
```

allocates 4 mega words, i.e., 16 mega bytes, to the control stack. You can use the parameter '-b' to specify the amount allocated to the trail stack, '-p' to the program area, and '-t' to the table area.

9.2 Garbage collection

B-Prolog incorporates an incremental garbage collector for the control stack and the heap. The garbage collector is active by default. It can be disabled by setting the Prolog flag `gc` to be `off`:

```
set_prolog_flag(gc,off)
```

The garbage collector is invoked automatically to reclaim the space taken by garbage in the top-most segment when the following condition is met:

$$H_{top} > C \times \frac{H_{avail}}{H_{max} - H_{avail}}$$

where H_{top} is the amount of memory in the heap top-segment, H_{avail} is the amount of available heap space, H_{max} is the total amount of space allocated to the heap, and C is a constant. GC will be invoked more frequently with the amount of heap space becoming less and less compared with the amount of consumed space. The threshold constant `C` is set to be `1000` bytes as a default. The user can reset the value by using the following primitive.

```
set_prolog_flag(gc_threshold,Value)
```

After the call, `Value` will become the new threshold constant.

The user can start the garbage collector explicitly by calling the following built-in predicate:

```
garbage_collect
```

and can check the number of garbage collections that have been performed since the system was started by using `statistics/0` or `statistics/2`.

Chapter 10

Matching Clauses

Matching clauses comprise a language for writing determinate Prolog programs. A matching clause takes the following form:

```
    H, G  => B
```

where `H` is an atomic formula, `G` and `B` are two sequences of atomic formulas. `H` is called the head, `G` the guard, and `B` the body of the clause. No call in `G` can bind variables in `H` and all calls in `G` must be in-line. In orther words, the guard must be *flat*. The following types of predicates can occur in `G`:

- Type checking
  - `integer(X)`,`real(X)`,`float(X)`,`number(X)`,`var(X)`,`nonvar(X)`,`atom(X)`, `atomic(X)`: `X` must be a variable that occurs before in either the head or some other call in the guard.
- Matching
  - `X=Y`: One of the arguments must be a non-variable term and the other must be a variable that occurs before. The non-variable term serves as a pattern and the variable refers to an object to be matched against the pattern. This call succeeds when the pattern and the object become identical after a substitution is applied to the pattern. For instance, the call `f(X)=Y` in a guard succeeds when `Y` is a structure whose functor is `f/1`.
- Term inspection
  - `functor(T,F,N)`: `T` must be a variable that occurs before. The call succeeds if the `T`'s functor is `F/N`. `F` can be either an atom or a variable. If `F` is not a first-occurrence variable, then the call is equivalent to `functor(T,F1,N),F1==F`. Similarly, `N` can be either an integer or a variable. If `N` is not a first-occurrence variable, then the call is equivalent to `functor(T,F,N1),N1==N`.

  - `arg(N,T,A)`: `T` must be a variable that occurs before and `N` must be an integer that is in the range of `1` and the arity of `T`, inclusive. If `A` is a first-occurrence variable, the call succeeds and binds `A` to the `N`th argument of `T`. If `A` is a variable that occurs before, the call is equivalent to `arg(N,T,A1),A1==A`. If `A` is a non-variable term, then the call is equivalent to `arg(N,T,A1),A1=A` where `A` is a pattern and `A1` is an object to be matched against `A`.
  - `T1 == T2`: `T1` and `T2` are identical terms.
  - `T1 \== T2`: `T1` and `T2` are not identical terms.

- Arithmetic comparisons
  - `E1 =:= E2`,`E1 =\= E2`, `E1 > E2`, `E1 >= E2`, `E1 < E2`, `E1 =< E2`: `E1` and `E2` must be ground expressions.

For a call `C`, matching rather than unification is used to select a matching clause in its predicate. The matching clause `H, G => B` is applicable to `C` if `C` matches `H` (i.e., `C` and `H` become identical after a substitution is applied to `H`) and `G` succeeds. When applying the matching clause to `C`, the system rewrites `C` into `B`.

Example:

```
membchk(X,[X|_]) => true.
membchk(X,[_|Ys]) => membchk(X,Ys).
```

This predicate checks whether or not an element given as the first argument occurs in a list given as the second argument. The head of the first clause `membchk(X,[X|_])` matches any call whose first argument is identical to the first element of the list. For instances, the calls `membchk(a,[a])` and `membchk(X,[X,Y])` succeed, and the calls `membchk(a,Xs)`, `membchk(a,[X])` and `membchk(X,[a])` fail.

Example:

```
append([],Ys,Zs) => Zs=Xs.
append([X|Xs],Ys,Zs) => Zs=[X|Zs1],append(Xs,Ys,Zs1).
```

This predicate concatenates two lists given as the first two arguments and returns the concatenated list through the third argument. Notice that all output unifications that bind variables in heads must be moved to the right hand sides of clauses. In comparison with the counterpart in standard Prolog clauses, this predicate cannot be used to split a list given as the third argument. In fact, the call `append(Xs,Ys,[a,b])` fails since it matches neither head of the clauses.

Matching clauses are determinate and employ one-directional matching rather than unification in the execution. The compiler takes advantage of these facts to generate more compact and faster code for matching clauses. While the compiler generates indexing code for Prolog clauses on at most one argument, it generates indexing code on as many arguments as possible. A program in matching clauses can be significantly faster than its counterpart in standard clauses if multi-level indexing is effective.

When consulted into the program code area, matching clauses are transformed into Prolog clauses that preserve the semantics of the original clauses. For example, after being consulted the `membchk` predicate becomes:

```
membchk(X,Ys):- $internal_match([Y|_],Ys),X==Y,!.
membchk(X,Ys):-$internal_match([_|Ys1],Ys),membchk(X,Ys1).
```

Where the predicate `$internal_match(P,O)` matches the object `O` against the pattern `P`.

Chapter 11

Action Rules and Events

Action rules comprise a programming construct that facilitates the description of reactive agents. An action rule specifies a pattern for agents, an action that the agents can carry out, and an event pattern for events that can activate the agents. An agent is a call or subgoal that is reactive to events. Agents are a more general notion than freeze in Prolog-II and processes in concurrent logic programming in the sense that the agents can be responsive to various kinds of events including user-defined ones. This chapter describes the syntax and semantics of action rules. Examples will be given in later chapters on the use of action rules to program constraint propagators and interactive user interfaces.

11.1 Syntax

An action rule, which extends a matching clause, takes the following form:

```
H, G, {E}  => B
```

The difference between an action rule and a matching clause is that an action rule has an event pattern `E` enclosed in a pair of braces after the guard.

There are a set of built-in events provided for programming constraint propagators, time-related behavior, and interactive graphical user interfaces. For example:

- `ins(X)` posted when the variable `X` is instantiated.
- `minmax(X)` posted when the bound of the domain variable `X` is updated.
- `dom(X,E)` posted when an inner element `E` is excluded from the domain of `X`.
- `time(T)` posted by the timer `T` every time a time interval elapses[1].

A user can create and post his/her own events and define agents to handle them. A user-defined event takes the form of `event(X,T)` where `X` is a variable, called a *suspension variable*, that connects the event with its handling agents, and `T` is a

[1]Timers work on Windows only now

Prolog term that contains the information to be transmitted to the agents. If the event poster does not have any information to be transmitted to the agents, then the second argument `T` can be omitted.

For example, the following defines an agent that echoes the messages sent to it by event posters.

```
echo_agent(X), {event(X,Message)} => write(Message).
```

Events are posted in most cases by built-ins, but a user's program can also post events by using the primitive `post(Event)`.

- `post(event(X,T))` Post an event `event(X,T)` to activate the agents on the suspension variable `X`. `T` carries some extra information to be transmitted to the agents.
- `post(event(X))` Equivalent to `post(event(X,[]))`.

For instance, the following query,

```
echo_agent(X), post(event(X,hello))
```

outputs the message `hello`. Notice that the event `event(X,hello)` will be ignored if no agent is waiting for it. Therefore, for the following query,

```
post(event(X,hello)), echo_agent(X)
```

no output will be given since the event is posted before the agent is created.

A predicate definition that contains at least one action rule is called an *agent definition*. Action rules and matching clauses can be mixed in agent definitions, but no standard Prolog clause is allowed.

11.2 Operational semantics

An action rule `H,G,{E} => B` is said to be applicable to an agent `C` if `C` matches `H` and the guard `G` succeeds. For an agent, the system searches for an applicable rule in its definition sequentially from the top. If no applicable rule is found, the agent fails; if a matching clause is found, then the agent is rewritten to the body of the clause as described before; if an action rule is found, then the agent is suspended, waiting until an event `E` is posted. When an event `E` is posted, the conditions in the guard are tested again. If they are satisfied, then the body `B` is executed. `B` is determinate in the sense that it cannot succeed more than once. When `B` fails, the original agent fails as well. After `B` is executed, the agent does not vanish but instead turns to wait until next event is posted.

Agents behave in an event-driven fashion. At the entry and exit points of every predicate, the system checks to see whether there is an event that has been posted. If so, then the current predicate is interrupted and control is moved to the lists of agents associated with the event. After the agents finish their execution, the interrupted predicate will resume. So, for the following query:

```
echo_agent(X),post(event(X,ping)),write(pong)
```

the output message will be `ping` followed by `pong`. The execution of `write(pong)` is interrupted after the event `event(X,ping)` is posted. The execution of agents can be further interrupted by other postings of events.

Agents are determinate in the sense that they can never succeed more than once. The system enforces this by removing all choice points created by the bodies of the rules in agent definitions. Consider the following program,

```
p(X,Y), var(X), {ins(X)} => true.
p(X,Y) => q(Y),write(Y).
q(1).
q(2).
```

and the query `p(a,Y),fail`. The action rule is not applicable to the agent `p(a,Y)` since `a` is not a variable. Therefore, the second rule is applied. Since no body can succeed more than once, the alternative clause `q(2)` will be cut off after the body succeeds with the output `1`. Notice that if the action rule is missing, then both `1` and `2` will appear in the output.

At a point during execution, there may be multiple events posted that are expected by an agent. If this is the case, then the agent has to be activated once for each of the events. The posting of an event may activate multiple agents. In the implementation, the agents that are created earlier are executed before those that are created later. Consider, for example, the following definition,

```
echo_agent(N,X), {event(X,Message)} => write(m(N,Message)).
```

and the query

```
echo_agent(1,X), acho_agent(2,X), post(event(X,hello)).
```

The output will be `m(1,hello)m(2,hello)`.

There is no primitive for killing agents explicitly. As described above, an agent never disappears as long as action rules are applied to it. An agent vanishes only when a matching clause is applied to it. Consider the following example.

```
echo_agent(X,Flag), var(Flag), {event(X,Message)} =>
    write(Message),Falg=1.
echo_agent(X,Flag) => true.
```

An echo agent defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. So, when a second event is posted, the action rule is no longer applicable and thus the matching clause after it will be selected. Notice that the matching clause is necessary here. Without it, an agent would fail after a second event is posted.

11.3 Timers

In some applications, agents are activated regularly at a predefined rate. For example, a clock animator is activated every second and the scheduler in a time-sharing system switches control to the next process after a certain time quata elapeses. To facilitate the description of time-related behavior of agents, B-Prolog provides timers[2]. To create a timer, use the predicate

```
timer(T,Interval)
```

where `T` is a variable and `Interval` is an integer that specifies the rate of the timer. A timer runs as a separate thread. The call `timer(T,Interval)` binds `T` to a Prolog term that represents the thread. A timer must be started first before use. To start a timer, use the call `timer_start(T)`. After started, the timer `T` posts an event `time(T)` in every `Interval` milliseconds. A timer stops posting events after the call `timer_stop(T)`. A stopped timer can be started again. A timer is destroyed after the call `timer_kill(T)` is executed.

- `timer(T,Interval)`: `T` is a timer with the rate being set to `Interval`.
- `timer(T)`: Equivalent to `timer(T,200)`.
- `timer_start(T)`: Start the timer `T`.
- `timer_stop(T)`: Stop the timer.
- `timer_kill(T)`: Kill the timer.
- `timer_set_interval(T,Interval)`: Set the interval of the timer `T` to `Interval`. The update is destructive and the old value is not restored upon backtracking.

Example:

The following example shows two agents that behave in accordance with two timers.

```
go:-
    timer(T1,100), timer_start(T1),
    timer(T2,1000),timer_start(T2),
    ping(T1),
    pong(T2),
    repeat,fail.


ping(T),{time(T)} => write(ping),nl.
pong(T),{time(T)} => write(pong),nl.
```

[2]In the current implementation, timers work on Windows only.

Notice that the two calls `repeat,fail` are needed after the two agents are created. Without them, the query `go` would succeed before any time event is posted and thus neither of the agent could get a chance to be activated.

11.4 Suspension and attribute variables

A suspension variable is a variable to which there are suspended agents and some other information attached. It is possible to attach any term to a suspension variable. Suspension variables are similar to attribute variables in some CLP systems and are useful for implementing user-defined domains such as rational numbers, sets, and floating-point intervals.

- `susp_attach_term(X,T)` Attach the term `T` to the variable `X`, where `X` must be a variable. The formerly attached term to `X`, if any, will be lost after this operation. This operation is undone automatically upon backtracking. In other words, the originally attached term will be restored upon backtracking.

- `susp_attached_term(X,T)` The currently attached term to the variable `X` is `T`.
- `frozen(L)` The list of all suspended agents is `L`.
- `frozen(V,L)` The list of suspended agents on the suspension variable `V` is `L`.

- `constraints_number(X,N)` `N` is the number of agents attached to the suspension variable `X`.

Example:

The following example shows how to attach a finite-domain to a variable:

```
create_fd_variable(X,D):-
    susp_attach_term(X,D),
    check_value(X,D).

check_value(X,D),var(X),{ins(X)} => true.
check_value(X,D) => member(X,D).
```

The agent `check_value(X,D)` is activated to check whether the value is in the domain when `X` is instantiated.

Chapter 12

Constraints

B-Prolog supports constraints over four different domains: finite-domains, Boolean, trees, and finite sets. The symbol `#=` is used to represent equality and `#\=` is used to represent inequality for all the three domains. The system decides what solver to call at run-time based on the type of the arguments.

12.1 CLP(Tree)

- `freeze(X,Goal)` Equivalent to `once(Goal)` but the evaluation is delayed until `X` becomes a non-variable term. The predicate is defined as follows:

  ```
  freeze(X,Goal),var(X),{ins(X)} => true.
  freeze(X,Goal) => call(Goal).
  ```

  If `X` is a variable, the agent `freeze(X,Goal)` is delayed. When `X` is bound, an event `ins(X)` is posted automatically, which will in turn activate the agent `freeze(X,Goal)`. If `X` is not a variable, then the second rule will rewrite `freeze(X,Goal)` into `call(Goal)`. Notice that since agents can never succeed more than once, `Goal` in `freeze(X,Goal)` cannot return multiple solutions. This is a big difference from the `freeze` predicate in Prolog-II.

- `dif(T1,T2)` The two terms `T1` and `T2` are different. If `T1` and `T2` are not arithmetic expressions, the constraint can be written as `T1 #\= T2`.

12.2 CLP(FD)

CLP(FD) is an extension of Prolog that supports built-ins for specifying domain variables, constraints, and strategies for instantiating variables. In general, a CLP(FD) program is made of three parts: the first part, called *variable generation*, generates variables and specifies their domains; the second part, called *constraint generation*, specifies constraints over the variables; and the final part, called *labeling*, instantiates the variables by doing enumeration.

Consider the well-known SEND MORE MONEY puzzle. Given eight letters S, E, N, D, M, O, R and Y, one is required to assign a digit between 1 and 9 to each letter such that different letters are assigned unique different digits and the equation SEND + MORE = MONEY holds. The following program specifies the problem.

```
sendmory(Vars):-
   Vars=[S,E,N,D,M,O,R,Y], % variable generation
   Vars :: 0..9,
   alldifferent(Vars),     % constraint generation
   S #\= 0,
   M #\= 0,
             1000*S+100*E+10*N+D
           + 1000*M+100*O+10*R+E
   #= 10000*M+1000*O+100*N+10*E+Y,
   labeling(Vars).         % labeling
```

The call `alldifferent(Vars)` ensures that variables in the list `Vars` take different values, and `labeling(Vars)` instantiates the list of variables `Vars` in the given order from left to right.

12.2.1 Finite-domain variables

A *finite domain* is a list of different *ground* terms in the form: `[`e_1,e_2,...,e_n`]`. The special notation `L..U` denotes a set of integers between L and U, inclusive.

A Prolog variable `Var` becomes a *finite-domain variable* after its domain is declared by:

```
Var :: D
```

or

```
Var in D
```

where `D` is a finite-domain. For example, the call

```
X :: 1..3
```

says that `X` is an integer between 1 and 3 and the call

```
X :: [a,b,c]
```

says that `X` can be `a`, `b`, or `c`.

Let `Vars` be a list of variables that share the same domain `D`. The domain of the variables can be declared as follows:

```
Vars :: D
```

If the domain is the set of integers between `L` and `U`, then the domain can also be declared as

```
domain(Var,L,U)
```

for one variable `Var` and as

```
domain(Vars,L,U)
```

for a list of variables `Vars`.

The following primitives restrict the domains of variables.

- `domain(Var,L,U)`: The domain of the variable `Var` is the set of integers between `L` and `U`. `L` and `U` must be integers. If `Var` is an integer, the call is equivalent to

  ```
  Var>=L,Var=<U
  ```

  If `Var` is neither a variable nor an integer, this call raises the `illegal_argument` exception.

- `domain([V1,...,Vn],L,U)`: Equivalent to:

  ```
  domain(V1,L,U),...,domain(Vn,L,U)
  ```

- `Var :: D`: The domain of the variable `Var` is `D`, where `D` is either an integer interval `L..U` or a list of ground terms. If `D` is a list that contains non-integer terms, then the call is equivalent to `member(Var,D)`.

- `[V1,...,Vn] :: D`: Equivalent to:

  ```
  V1 :: D,...,Vn :: D
  ```

- `Var notin D`: `Var` does not reside in `D`.

- `[V1,...,Vn] notin D`: Equivalent to:

  ```
  V1 noin D,...,Vn noin D
  ```

The following primitives are available on integer domain variables. As domain variables are also suspension variables, primitives on suspension variables such as `frozen/1` can be applied to domain variables as well.

- `fd_var(V)` `V` is a domain variable.

- `fd_max(V,N)` The maximum element in the domain of `V` is `N`. `V` must be an integer domain variable or an integer.

- `fd_min(V,N)` The minimum element in the domain of `V` is `N`. `V` must be an integer domain variable or an integer.

- `fd_min_max(V,Min,Max)` The minimum and maximum elements in the domain of `V` are `Min` and `Max`, respectively. `V` must be an integer domain variable or an integer.

- `fd_size(V,N)` The size of the domain of `V` is `N`.

- `fd_dom(V,L)` `L` is the list of elements in the domain of `V`.

- `fd_true(V,E)` `E` is an element in the domain of `V`.

- `fd_next(V,E,NextE)` `NextE` is the next element following `E` in `V`'s domain.

- `fd_prev(V,E,PrevE)` `PrevE` is the element preceding `E` in `V`'s domain.

- `fd_include(V1,V2)` Succeeds if `V1`'s domain includes `V2`'s domain as a set.

- `fd_vector_min_max(Min,Max)` Specifies the range of bit vectors. Domain variables, when being created, are usually represented internally by using intervals. An interval turns to a bit vector when a hole occurs in it. The default values for `Min` and `Max` are -320 and 320, respectively.

12.2.2 Arithmetic constraints

An arithmetic constraint is a call in the form,

```
E1 R E2
```

where `E1` and `E2` are two arithmetic expressions and `R` is one of the following constraint symbols `#=`, `#\=`, `#>=`, `#>`, `#=<`, and `#<`. An arithmetic expression is made of integers, variables, domain variables, and the following arithmetic functions: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `**` (power), `abs`, `min`, `max`, and `sum`. The `**` operator has the highest priority, followed by `*` and `/`, then followed by unary minus sign `-`, and finally followed `+` and `-`.

Let `E` be an expression, and `L` be a list of expressions `[E1,E2,...,En]`. The following are valid expressions.

- `abs(E)` The absolute of `E`.

- `min(L)` The minimum element of `L`.

- `max(L)` The maximum element of `L`.

- `sum(L)` The sum of the elements of `L`.

12.2.3 Global constraints

- `alldifferent(Vars)`
- `all_different(Vars)` The elements in `Vars` are mutually different, where `Vars` is a list of terms.
- `alldistinct(Vars)`
- `all_distinct(Vars)` This is equivalent to `alldifferent(Vars)`, but it uses a stronger consistency checking algorithm to exclude inconsistent values from domains of variables. See the next chapter for the implementation of this constraint.
- `fd_element(I,L,V)`
- `element(I,L,V)` Succeeds if the `I`th element of `L` is `V`, where `I` must be an integer or an integer domain variable, `V` a term, and `L` a list of terms.
- `fd_atmost(N,L,V)`
- `atmost(N,L,V)` Succeeds if there are at most `N` elements in `L` that are equal to `V`, where `N` must be an integer, `V` a term, and `L` a list of terms.
- `cumulative(Starts,Durations,Resources,Limit)` This constraint is useful for describing and solving scheduling problems. The arguments `Starts`, `Durations`, and `Resources` are lists of integer domain variables of the same length and `Limit` is an integer domain varialable. Let `Starts` be `[S1,S2,...,Sn]`, `Durations` be `[D1,D2,...,Dn]` and `Resources` be `[R1,R2,...,Rn]`. For each job `i`, `Si` represents the start time, `Di` the duration, and `Ri` the units of resources needed. `Limit` is the units of resources available at any time.
- `diffn(L)` This constraint ensures that no two rectangles in `L` overlap with each other. A rectangle in a n-dimensional space is represented by a list of $2 \times n$ elements `[X1,X2,...,Xn,S1,S2,...,Sn]` where `Xi` is the starting coordinate of the edge in the ith dimension and `Si` is the size of the edge.

12.2.4 Labeling and variable ordering

Several predicates are provided for choosing variables and assigning values to variables.

- `indomain(V)` `V` is instantiated to a value in the domain. On backtracking, the domain variable is instantiated to the next value in the domain.
- `deleteff(V,Vars,Rest)` Chooses first a domain variable `V` from `Vars` with the minimum domain. `Rest` is a list of domain variables without `V`.
- `deleteffc(V,Vars,Rest)` Chooses first a variable that has the smallest domain and that participates in the largest number of constraints.

- `labeling(Vars)`
- `fd_labeling(Vars)` Instantiates the variables in `Vars` one by one.
- `labeling_ff(Vars)`
- `labelingff(Vars)`
- `fd_labeling_ff(Vars)` Instantiates the variables in `Vars` by using `deleteff/3` to choose variables.
- `labeling_ffc(Vars)`
- `labelingffc(Vars)`
- `fd_labeling_ffc(Vars)` Instantiates the variables in `Vars` by using `deleteffc/3` to choose variables.

12.2.5 Optimization

- `fd_minimize(Goal,Exp)` This primitive finds a satisfiable instance of `Goal` such that `Exp` has the minimum value. Here, `Goal` is used as a generator (e.g., `labeling(L)`), and `Exp` is an expression. All satisfiable instances of `Goal` must be ground, and for every such instance, `Exp` must be an integer expression.
- `fd_maximize(Goal,Exp)` This primitive finds a satisfiable instance of `Goal` such that `Exp` has the maximum optimal value. It is equivalent to `fd_minimize(Goal,-Exp)`.

12.3 CLP(Boolean)

CLP(Boolean) can be considered as a special case of CLP(FD) where each variable has a domain of two values: 0 denotes *false*, and 1 denotes *true*. A Boolean expression is made from constants (0 or 1), Boolean domain variables, basic relational constraints, and operators as follows:

```
<BooleanExpression> ::=
     0 |                                              /* false */
     1 |                                              /* true */
     variable |
     <Expression> #= <Expression> |
     <Expression> #\= <Expression> |
     <Expression> #> <Expression> |
     <Expression> #>= <Expression> |
     <Expression> #< <Expression> |
     <Expression> #=< <Expression> |
     #\ <BooleanExpression> |                          /* not */
```

```
<BooleanExpression> #/\ <BooleanExpression> |  /* and */
<BooleanExpression> #\/ <BooleanExpression> |  /* or */
<BooleanExpression> #=> <BooleanExpression> |  /* imply */
<BooleanExpression> #<=> <BooleanExpression> | /* equivalent */
<BooleanExpression> #\ <BooleanExpression>     /* not equal */
```

A Boolean constraint is made of a constraint symbol and one or two Boolean expressions.

- `E1 #= E2` True if `E1` and `E2` are equivalent. For example, `(X #= 3) #= (Y #= 5)` means that the finite-domain constraints `(X #= 3)` and `(Y #= 5)` have the same satisfibility. In other words, they are either both true or both false.
- `E1 #\= E2` True if `E1` and `E2` are different. For example, `(X #= 3) #\= (Y #= 5)` means that the finite-domain constraints `(X #= 3)` and `(Y #= 5)` are mutually exclusive. In other words, if `(X #= 3)` is satisfied then `(Y #= 5)` cannot be satisfied, and similarly if `(X #= 3)` is not satisfied then `(Y #= 5)` must be satisfied.
- `#\ E` Equivalent to `E#=0`.
- `E1 #/\ E2` Both `E1` and `E2` are 1.
- `E1 #\/ E2` Either `E1` or `E2` is 1.
- `E1 #=> E2` If `E1` is 1, then `E2` must be also 1.
- `E1 #<=> E2` `E1` and `E2` are equivalent.
- `E1 #\ E2` Exactly one of `E1` and `E2` is 1.

The following constraints restrict the values of Boolean variables.

- `fd_at_least_one(L)`
- `at_least_one(L)` Succeeds if at least one element in L is equal to 1, where L is a list of Boolean variables or constants.
- `fd_at_most_one(L)`
- `at_most_one(L)` Succeeds if at most one element in L is equal to 1, where L is a list of Boolean variables or constants.
- `fd_only_one(L)`
- `only_one(L)` Succeeds if exactly one element in L is equal to 1, where L is a list of Boolean variables or constants.

12.4 CLP(Set)

CLP(Set) is a member in the CLP family where each variable can have a set as its value. Although a number of languages are named CLP(Set), they are quite different. Some languages allow intentional and infinite sets, and some languages allows user-defined function symbols in set constraints. The CLP(Set) language in B-Prolog allows finite sets of ground terms only. A *set constant* is either the empty set `{}` or `{T1,T2,...,Tn}` where each `Ti` (i=1,2,...,n) is a ground term.

We reuse some of the operators in Prolog and CLP(FD) (e.g., `/\`, `\/`, `\`, `#=`, and `#\=`) and introduce several new operators to the language to denote set operations and set constraints. Since most of the operators are generic and their interpretation depends on the types of constraint expressions, the users have to provide necessary information for the system to infer the types of expressions.

The type of a variable can be known from its domain declaration or can be inferred from its context. The domain of a set variable is declared by a call as follows:

```
V :: L..U
```

where `V` is a variable, and `L` and `U` are two set constants indicating respectively the lower and upper bounds of the domain. The lower bound contains all *definite* elements that are known to be in `V` and the upper bound contains all *possible* elements that may be in `V`. All definite elements must be possible. In other words, `L` must be a subset of `U`. If this is not the case, then the declaration fails. The special set constant `{I1..I2}` represents the set of integers in the range from `I1` to `I2`, inclusive. For example:

- `V :: {}..{a,b,c}` : *V* is subset of `{a,b,c}` including the empty set.
- `V :: {1}..{1..3}` : `V` is one of the sets of `{1}`,`{1,2}`, `{1,3}`, and `{1,2,3}`. The set `{2,3}` is not a candidate value for `V`.
- `V :: {1}..{2,3}` : Fails since `{1}` is not a subset of `{2,3}`.

We extend the notation such that `V` can be a list of variables. So the call

```
[X,Y,Z] :: {}..{1..3}
```

declares three set variables.

The following primitives are provided to test and access set variables:

- `clpset_var(V)`: `V` is a set variable.
- `clpset_low(V,Low)`: The current lower bound of `V` is `Low`.
- `clpset_up(V,Up)`: The current upper bound of `V` is `Up`.
- `clpset_added(E,V)`: `E` is a definite element, i.e., an element included in the lower bound.

- `clpset_excluded(E,V)`: `E` has been forbidden for `V`. In other words, `E` has been excluded from the upper bound of `V`.

The followint two predicates are provided for converting sets into and from lists:

- `set_to_list(S,L)` : Convert the set S into a list. For example,
- `list_to_set(L,S)` : Convert the list L into a set.

A *set expression* is defined recursively as follows: (1) a constant set; (2) a variable; (3) a composite expression in the form of `S1 \/ S2`, `S1 /\ S2`, `S1 \ S2`, or `\ S1`, where `S1` and `S2` are set expressions. The operators `\/` and `/\` represent union and intersection, respectively. The binary operator `\` represents difference and the unary operator `\` represents complement. The complement of a set `\ S1` is equivalent to `U \ S1` where `U` is the universal set. Since the universal set of a constant is unknown, `S1` in the expression `\ S1` must be a variable whose universal set has been declared.

We extend the syntax for finite-domain constraint expressions to allow the expression `#S` which denotes the cardinality of the set represented by the set expression `S`.

Let `S`, `S1` and `S2` be set expressions, and `E` be a term. A set constraint takes one of the following forms:

- `S1 #= S2`: `S1` and `S2` are two equivalent sets ($S1=S2$).
- `S1 #\= S2`: `S1` and `S2` are two different sets ($S1\neq S2$).
- `S1 subset S2`: `S1` is a subset of `S2` ($S1\subseteq S2$). The proper subset relation $S1 \subset S2$ can be represented as `S1 subset S2` and `#S1 #< #S2` where `#<` represents the less-than constraint on integers.
- `S1 #<> S2`: `S1` and `S2` are disjoint ($S1\cap S2=\emptyset$).
- `E #<- S`: `E` is a member of `S` ($E\in S$).
- `E #<\- S`: `E` is a not member of `S` ($E\notin S$).

Boolean constraint expressions are extended to allow set constraints. For example, the constraint

```
(E #<- S1) #=> (E #<- S2)
```

says that if `E` is a member of `S1` then `E` must also be a member of `S2`.

Just as for finite and Boolean constraints, constraint propagation is used to maintain the consistency of constraints. Constraint propagation alone, however, is inadequate for finding solutions for many problems. We need to use the *divide-and-conquer* or *relaxation* method to find solutions to a system of constraints. The call

- `indomain(V)`

finds a value for `V` either by enumerating the values in `V`'s domain or by splitting the domain. Instantiating variables usually triggers related constraint propagators.

Chapter 13

Programming Constraint Propagators

Action rules form a powerful implementation language for programming constraint propagators. We will show in this chapter how to program constraint propagators for various constraints.

The following set of event types are provided for programming constraint propagators:

- `ins(X)`: `X` or any variable in `X` is instantiated.
- `minmax(X)`: The low or upper bound of any variable in `X` is updated.
- `dom(X)`: An inner element has been excluded from the domain of `X`.
- `dom(X,E)`: An inner element `E` has been excluded from the domain of `X`.

Except for the last event type that has two arguments, all the events do not have extra information to be transmitted to their handlers. For these types of events, we extend action rules to enable them to carry multiple events. An agent defined by this type of rule is activated when one of the events is posted. For example, for the following rule,

```
    p(X),{ins(X),minmax(X)} => q(X).
```

`p(X)` is activated when `X` is instantiated, or either bound of `X`'s domain is updated.

We also introduce the following two types of conditions that can occur in the guards of rules:

- `dvar(X)`: `X` is an integer domain variable.
- `no_vars_gt(m,n)`: The number of variables in the last `m` arguments of the agent is greater than `n`. Notice that the condition does not take the arguments whose variables are to be counted. The system can always fetch that information from its parent call. This condition cannot be used in matching clauses.

13.1 A constraint interpreter

It is very easy to write a constraint interpreter by using action rules. The following shows such an interpreter:

```
      interp_constr(Constr), no_vars_gt(1,0),
            {ins(Constr),minmax(Constr)}
            =>
           reduce_domains(Constr).
      interp_constr(Constr) => test_constr(Constr).
```

For a constraint `Constr`, if there is at least one variable in it, the interpreter delays the constraint and invokes the procedure `reduce_domains(Constr)` to exclude no-good values from the variables in `Constr`. The two kinds of events, namely `ins(Constr)` and `minmax(Constr)` ensure that the constraint will be reconsidered whenever either a bound of a variable in `Constr` is updated or a variable is bound to any value.

13.2 Indexicals

Indexicals, which are adopted by many CLP(FD) compilers for compiling constraints, can be implemented easily by using action rules. Consider the indexical

```
      X in min(Y)+min(Z)..max(Y)+max(Z).
```

which ensures that the constraint `X #= Y+Z` is interval-consistent on `X`. The indexical is activated whenever a bound of `Y` or `Z` is updated. The following shows the implementation in action rules:

```
    'V in V+V'(X,Y,Z),{ins(Y),minmax(Y),ins(Z),minmax(Z)} =>
          reduce_domain(X,Y,Z).

    reduce_domain(X,Y,Z) =>
          fd_min_max(Y,MinY,MaxY),
          fd_min_max(Z,MinZ,MaxZ),
          L is MinY+MinZ, U is MaxY+MaxZ,
          X in L..U.
```

The action `reduce_domain(X,Y,Z)` is executed whenever a variable is instantiated or a bound of a variable is updated. The original indexical is equivalent to the following two calls:

```
    'V in V+V'(X,Y,Z),reduce_domain(X,Y,Z)
```

Interval-consistency is also enforced on `X` when the constraint is generated.

13.3 Reification

One well used technique in finite-domain constraint programming is called *reification*, which uses a new Boolean variable `B` to indicate the satisfiability of a constraint `C`. `C` must be satisfied if and only if `B` is equal to `1`. This relationship is denoted as:

```
C #<=> (B #= 1)
```

It is possible to use Boolean constraints to represent the relationship, but it is more efficient to implement specialized propagators to maintain the relationship. Consider, as an example, the reification:

```
(X #= Y) #<=> (B #= 1)
```

where `X` and `Y` are domain variables, and `B` is a Boolean variable. The following describes a propagator that maintains the relationship:

```
reification(X,Y,B),dvar(B),dvar(X),X\==Y,
     {ins(X),ins(Y),ins(B)} => true.
reification(X,Y,B),dvar(B),dvar(Y),X\==Y,
      {ins(Y),ins(B)} => true.
reification(X,Y,B),dvar(B),X==Y => B=1.
reification(X,Y,B),dvar(B) => B=0.
reification(X,Y,B) => (B==0 -> X #\= Y; X #= Y).
```

Curious readers might have noticed that `ins(Y)` is in the event sequence of the first rule but `ins(X)` is not specified in the second one. The reason for this is that `X` can never be a variable after the condition of the first rule fails and that of the second rule succeeds.

13.4 Propagators for binary constraints

There are different levels of consistency for constraints. A unary constraint `p(X)` is said to be *domain-consistency* if for any element `x` in the domain of `X` the constraint `p(x)` is satisfied. The propagation rule that maintains domain-consistency is called *forward checking*. A constraint is said to be *interval-consistent* if for any bound of the domain of any variable there are supporting elements in the domains of the all other variables such that the constraint is satisfied. Propagators for maintaining interval consistency are activated whenever a bound of a variable is updated or whenever a variable is instantiated. A constraint is said to be *arc-consistent* if for any element in the domain of any variable there are supporting elements in the domains of all the other variables such that the constraint is satisfied. Propagators for maintaining domain consistency are triggered when whatever changes occur to the domain of a variable. We consider how to implement various propagators for the binary constraint `A*X #= B*Y+C`, where `X` and `Y` are domain variables, `A` and `B` are positive integers, and `C` is an integer of any kind.

Forward checking

The following shows a propagator that performs forward checking for the binary constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_forward'(A,X,B,Y,C).

'aX=bY+c_forward'(A,X,B,Y,C),var(X),var(Y),{ins(X),ins(Y)} => true.
'aX=bY+c_forward'(A,X,B,Y,C),var(X) =>
       T is B*Y+C, Ex is T//A, (A*Ex=:=T->X = Ex; true).
'aX=bY+c_forward'(A,X,B,Y,C) =>
       T is A*X-C, Ey is T//B, (B*Ey=:=T->Y is Ey;true).
```

When both X and Y are variables, the propagator is suspended. When either variable is instantiated, the propagator computes the value for the other variable.

Interval-consistency

The following propagator, which extends the forward-checking propagator, maintains interval-consistency for the constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_reduce_domain'(A,X,B,Y,C),
      'aX=bY+c_forward'(A,X,B,Y,C),
      'aX=bY+c_interval'(A,X,B,Y,C).
```

The call `'aX=bY+c_reduce_domain'(A,X,B,Y,C)` preprocess the constraint to make it interval-consistent when the constraint is generated.

```
'aX=bY+c_reduce_domain'(A,X,B,Y,C) =>
      'aX in bY+c_reduce_domain'(A,X,B,Y,C),
      MC is -C,
      'aX in bY+c_reduce_domain'(B,Y,A,X,MC).

'aX in bY+c_reduce_domain'(A,X,B,Y,C) =>
      L is (B*min(Y)+C) /> A,
      U is (B*max(Y)+C) /< A,
      X in L..U.
```

The operation `op1 /> op2` returns the lowest integer that is greater than or equal to the quotient of `op1` by `op2` and the operation `op1 /< op2` returns the greatest integer that is less than or equal to the quotient. The arithmetic operations must be sound to make sure that no solution is lost. For example, the minimum times any positive integer remains the minimum.

The call `'aX=bY+c_interval'(A,X,B,Y,C)` maintains interval-consistency for the constraint.

```
'aX=bY+c_interval'(A,X,B,Y,C) =>
      'aX in bY+c_interval'(A,X,B,Y,C),  % reduce X when Y changes
      MC is -C,
      'aX in bY+c_interval'(B,Y,A,X,MC). % reduce Y when X changes

'aX in bY+c_interval'(A,X,B,Y,C),var(X),var(Y),{minmax(Y)} =>
      'aX in bY+c_reduce_domain'(A,X,B,Y,C).
'aX in bY+c_interval'(A,X,B,Y,C) => true.
```

Notice that the action `'aX=bY+c_reduce_domain'(A,X,B,Y,C)` is executed only when both variables are free. If either one turns to be instantiated, then the forward-checking rule will take care of that situation.

Arc-consistency

The following propagator, which extends the one shown above, maintains arc-consistency for the constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_reduce_domain'(A,X,B,Y,C),
      'aX=bY+c_forward'(A,X,B,Y,C),
      'aX=bY+c_interval'(A,X,B,Y,C),
      'aX=bY+c_arc'(A,X,B,Y,C).

 'aX=bY+c_arc'(A,X,B,Y,C) =>
      'aX in bY+c_arc'(A,X,B,Y,C),  % reduce X when Y changes
      MC is -C,
      'aX in bY+c_arc'(B,Y,A,X,MC). % reduce Y when X changes

 'aX in bY+c_arc'(A,X,B,Y,C),var(X),var(Y),{dom(Y,Ey)} =>
      T is B*Ey+C,
      Ex is T//A,
      (A*Ex=:=T -> exclude(X,Ex);true).
 'aX in bY+c_arc'(A,X,B,Y,C) => true.
```

Whenever an element `Ey` is excluded from the domain of `Y`, the propagator `'aX in bY+c_arc'(A,X,B,Y,C)` is activated. If both `X` and `Y` are variables, the propagator will exclude `Ex`, the counterpart of `Ey`, from the domain of `X`. Again, if either `X` or `Y` becomes an integer, the propagator does nothing. The forward checking rule will take care of that situation.

13.5 all_different(L)

The constraint `all_different(L)` holds if the variables in `L` are pair-wisely different. One naive implementation method for this constraint is to generate binary disequality constraints between all pairs of variables in `L`. This implementation has two problems: First, the space required to store the constraints is quadratic in the number of variables in L; Second, splitting the constraint into small granularity ones may lose possible propagation opportunities.

To solve the space problem, we define all_different(L) in the following way:

```
all_different(L) => all_different(L,[]).

all_different([],Left) => true.
all_different([X|Right],Left) =>
    outof(X,Left,Right),
    all_different(Right,[X|Left]).

outof(X,Left,Right), var(X), {ins(X)} => true.
outof(X,Left,Right) =>
    exclude_list(X,Left),exclude_list(X,Right).
```

For each variable X, let Left be the list of variables to the left of X and Right be the list of variables to the right of X. The call outof(X,Left,Right) holds if X appears in neither Left nor Right. Instead of generating disequality constraints between X and all the variables in Left and Right, the call outof(X,Left,Right) suspends until X is instantiated. After X becomes an integer, the calls exclude_list(X,Left) and exclude_list(X,Right) to exclude X from the domains of the variables in Left and Right, respectively.

There is a propagator outof(X,Left,Right) for each element X in the list, which takes constant space. Therefore, all_different(L) takes linear space in the size of L. Notice that the two lists Left and Right are not merged into one bigger list. Or, the constraint still takes quadratic space.

Chapter 14

External Language Interface with C

B-Prolog has a bi-directional interface with C through which Prolog programs can call functions written in C and C programs can call Prolog as well. C programs that use this interface must include the file `"bprolog.h"` in the directory `$BPDIR/Emulator`.

The functions are renamed in Version 6.0 such that all function names start with `''bp_''`. Old functions except for `build_LIST` and `build_STRUCTURE` are still supported but they are not documented here. The user is encouraged to use the new functions.

14.1 Calling C from Prolog

14.1.1 Term representation

A term is represented by a word containing a value and a tag. The tag distinguishes the type of the term. Floating-point numbers are represented as special structures in the form of `$float(I1,I2,I3)` where `I1`, `I2` and `I3` are integers.

The value of a term is an address except when the term is an integer (in this case, the value represents the integer itself). The address points to a different location depending on the type of the term. The address in a reference points to the referenced term. An unbound variable is represented by a self-referencing pointer. The address in an atom points to the record for the atom symbol in the symbol table. The address in a structure $f(t_1, \ldots, t_n)$ points to a block of $n+1$ consecutive words where the first word points to the record for the functor f/n in the symbol table and the remaining n words store the components of the structure. The address in a list $[H|T]$ points to a block of two consecutive words where the first word stores the car H and the second word stores the cdr T.

14.1.2 Fetching arguments of Prolog calls

Every C function that defines a Prolog predicate should not take any argument. The function `bp_get_call_arg(i,arity)` is used to get the arguments in the current Prolog call:

- `TERM bp_get_call_arg(int i, int arity)`
  Fetch the ith argument, where `arity` is the arity of the predicate, and `i` must be an integer between 1 and `arity`. The validness of the arguments are not checked and an invalid argument may cause fatal errors.

14.1.3 Testing Prolog terms

The following functions are provided for testing Prolog terms. They return `BP_TRUE` when succeed and `BP_FALSE` when fail.

- `int bp_is_atom(TERM t)` Term `t` is an atom.
- `int bp_is_integer(TERM t)` Term `t` is an integer.
- `int bp_is_float(TERM t)` Term `t` is a floating-point number.
- `int bp_is_nil(TERM t)` Term `t` is a nil.
- `int bp_is_list(TERM t)` Term `t` is a list.
- `int bp_is_structure(TERM t)` Term `t` is a structure (but not a list).
- `int bp_is_compound(TERM t)` True if either `bp_is_list(t)` or `bp_is_structure(t)` is true.
- `int bp_is_unifiable(TERM t1, TERM t2)` `t1` and `t2` are unifiable. This is equivalent to the Prolog call `not(not(t1=t2))`.
- `int bp_is_identical(TERM t1, TERM t2)` `t1` and `t2` are identical. This function is equivalent to the Prolog call `t1==t2`.

14.1.4 Converting Prolog terms into C

The following functions convert Prolog terms to C. If a Prolog term is not of the expected type, then the global C variable `exception` is set. A C program that uses these functions must check whether `exception` is set to see whether data are converted correctly. The converted data are correct only when `exception` is `NULL`.

- `int bp_get_integer(TERM t)` Convert the Prolog integer `t` into C. `bp_is_integer(t)` must be true; otherwise `exception` is set to `integer_expected` and 0 is returned.
- `double bp_get_float(TERM t)` Convert the Prolog float `t` into C. `bp_is_float(t)` must be true; otherwise `exception` is set to `number_expected` and 0.0 is returned. This function must be declared before any use.

- `(char *) bp_get_name(TERM t)` Return a pointer to the string that is the name of term `t`. Either `bp_is_atom(t)` or `bp_is_structure(t)` must be true; otherwise, `exception` is set to `illegal_arguments` and `NULL` is returned. This function must be declared before any use.
- `int bp_get_arity(TERM t)` Return the arity of term `t`. Either `bp_is_atom(t)` or `bp_is_structure(t)` must be true; otherwise, `exception` is set to `illegal_arguments` and 0 is returned.

14.1.5 Manipulating and writing Prolog terms

- `int bp_unify(TERM t1,TERM t2)` Unify two Prolog terms `t1` and `t2`. The result is `BP_TRUE` if the unification succeeds and `BP_FALSE` if fails.
- `TERM bp_get_arg(int i,TERM t)` Return the `i`th argument of term `t`. `bp_is_compound(t)` must be true and `i` must be an integer that is greater than 0 and no greater than `t`'s arity; otherwise, `exception` is set to `illegal_arguments` and the Prolog integer `0` is returned.
- `TERM bp_get_car(TERM t)` Return the car of the list `t`. `bp_is_list(t)` must be true; or `exception` is set to `list_expected` and the Prolog integer `0` is returned.
- `TERM get_cdr(TERM t)` Return the cdr of the list `t`. `bp_is_list(t)` must be true; or `exception` is set to `list_expected` and the Prolog integer `0` is returned.
- `void bp_write(TERM t)` Send term `t` to the current output stream.

14.1.6 Building Prolog terms

- `TERM bp_build_var()` Return an free Prolog variable.
- `TERM bp_build_integer(int i)` Return a Prolog integer whose value is `i`.
- `TERM bp_build_float(double f)` Return a Prolog float whose value is `f`.
- `TERM bp_build_atom(char *name)` Return a Prolog atom whose name is `name`.
- `TERM bp_build_nil()` Return a Prolog empty list.
- `TERM bp_build_list()` Return a Prolog list whose car and cdr are free variables.
- `TERM bp_build_structure(char *name, int arity)` Return a Prolog structure whose functor is `name`, arity is `arity`, and the arguments are all free variables.

14.1.7 Registering predicates defined in C

The following function registers a predicate defined by a C function.

```
insert_cpred(char *name, int arity, int (*func)())
```

The first argument is the predicate name, the second is the arity, and the third is the name of the function that defines the predicate. The function cannot take any argument. As described before, the function `bp_get_call_arg(i,arity)` is used to fetch arguments from the Prolog call.

For example, the following registers a predicate whose name is `"p"` and whose arity is 2.

```
extern int p();
insert_cpred("p", 2, p)
```

the C function's name does not need to be the same as the predicate name.

Predicates defined in C should be registered after the Prolog engine is initialized and before any call is executed. One good place for registering predicates is the `Cboot()` function in the file `cpreds.c`, which registers all the built-ins of B-Prolog.

Example:

Consider the Prolog predicate:

```
 :-mode p(+,?).
p(a,f(1)).
p(b,[1]).
p(c,1.2).
```

where the first argument is given and the second is unknown. The following steps show how to define this predicate in C and make it callable from Prolog.

Step 1 . Write a C function to implement the predicate. The following shows a sample:

```
#include "bprolog.h"

p(){
  TERM a1,a2,a,b,c,f1,l1,f12;
  char *name_ptr;

  /*   prepare Prolog terms */
  a1 = bp_get_call_arg(1,2);  /* first argument */
  a2 =  bp_get_call_arg(2,2); /* second argument */
  a = bp_build_atom("a");
  b = bp_build_atom("b");
```

```
    c = bp_build_atom("c");
    f1 = bp_build_structure("f",1);  /* f(1) */
    bp_unify(bp_get_arg(1,f1),bp_build_integer(1));
    l1 = bp_build_list();            /* [1] */
    bp_unify(bp_get_car(l1),bp_build_integer(1));
    bp_unify(bp_get_cdr(l1),bp_build_nil());
    f12 = bp_build_float(1.2);       /* 1.2 */

    /* code for the clauses */
    if (!bp_is_atom(a1)) return BP_FALSE;
    name_ptr = bp_get_name(a1);
    switch (*name_ptr){
    case 'a':
      return (bp_unify(a1,a) ? bp_unify(a2,f1) : BP_FALSE);
    case 'b':
      return (bp_unify(a1,b) ? bp_unify(a2,l1) : BP_FALSE);
    case 'c':
      return (bp_unify(a1,c) ? bp_unify(a2,f12) : BP_FALSE);
    default: return BP_FALSE;
    }
  }
```

Step 2 Insert the folloiwng two lines into `Cboot()` in `cpreds.c`:

```
    extern int p();
    insert_cpred("p",2,p);
```

Step 3 Recompile the system. Now, `p/2` is in the group of built-ins in B-Prolog.

14.2 Calling Prolog from C

To make Prolog predicates callable from C, one has to replace the `main.c` file in the emulator with a new file that starts his/her own application. The following function must be executed before any call to Prolog predicates is executed:

```
initialize_bprolog(int argc, char *argv[])
```

In addition, the environment variable `BPDIR` must be set correctly to the home directory where B-Prolog was installed. The function `initialize_bprolog()` allocates all the stacks used in B-Prolog, initializes them, and loads the byte code file `bp.out` into the program area. `BP_ERROR` is returned if the system cannot be initialized.

A query can be a string or a Prolog term, and a query can return one solution and multiple solutions as well.

- `int bp_call_string(char *goal)` This function executes the Prolog call as represented by the string `goal`. The return value is `BP_TRUE` if the call succeeds, `BP_FALSE` if the call fails, and `BP_ERROR` if an exception occurs. Examples:

  ```
  bp_call_string("load(myprog)")
  bp_call_string("X is 1+1")
  bp_call_string("p(X,Y),q(Y,Z)")
  ```

- `bp_call_term(TERM goal)` This function is similar to the previous one, but executes the Prolog call as represented by the term `goal`. While `bp_call_string` cannot return any bindings for variables, this function can return results through the Prolog variables in `goal`. Example:

  ```
  TERM call = bp_build_structure("p",2);
  bp_call_term(call);
  ```

- `bp_mount_query_string(char *goal)` Mount `goal` as the next Prolog goal to be executed.

- `bp_mount_query_string(TERM goal)` Mount `goal` as the next Prolog goal to be executed.

- `bp_next_solution()` Retrieve the next solution of the current goal. If no goal is mounted before this function, then the exception `illegal_predicate` will be raised and `BP_ERROR` will be returned as the result. If no further solution is available, the function returns `BP_FALSE`. Otherwise, the next solution is found.

Example:

This example program retrieves all the solutions of the query `member(X,[1,2,3])`.

```
#include "bprolog.h"

main(argc,argv)
int             argc;
char           *argv[];
{
  TERM query;
  TERM list0,list;
  int res;

  initialize_bprolog(argc,argv);
  /* build the list [1,2,3] */
  list = list0 = bp_build_list();
```

```
  bp_unify(bp_get_car(list),bp_build_integer(1));
  bp_unify(bp_get_cdr(list),bp_build_list());
  list = bp_get_cdr(list);
  bp_unify(bp_get_car(list),bp_build_integer(2));
  bp_unify(bp_get_cdr(list),bp_build_list());
  list = bp_get_cdr(list);
  bp_unify(bp_get_car(list),bp_build_integer(3));
  bp_unify(bp_get_cdr(list),bp_build_nil());

  /* build the call member(X,list) */
  query = bp_build_structure("member",2);
  bp_unify(bp_get_arg(2,query),list0);

  /* invoke member/2 */
  bp_mount_query_term(query);
  res = bp_next_solution();
  while (res==BP_TRUE){
    bp_write(query); printf("\n");
    res = bp_next_solution();
  }
}
```

To run the program, we need to first replace the content of the file `main.c` in `$BPDIR/Emulator` with this program and recompile the system. The newly compiled system will give the following outputs when started.

```
member(1,[1,2,3])
member(2,[1,2,3])
member(3,[1,2,3])
```

Chapter 15

External Language Interface with Java

As the popularity of Java grows, an interface that bridges Prolog and Java becomes more and more important. On the one hand, Prolog applications can have access to resources in Java, such as the Abstract Window Toolkit(AWT) and networking. On the other hand, Java programs can have access to the functionality such as constraint solving available in Prolog. B-Prolog has a bi-directional interface with Java, which is based on JIPL developed by Nobukuni Kino.

An application that uses the Java interface usually works as follows: The Java part invokes a Prolog predicate and passes it a Java object together with other arguments; the Prolog predicate performs necessary computation and invokes the methods or directly manipulates the fields of the Java object. The readers are recommended to see the examples at the directory `$BPDIR/Examples/java_interface` before writing their own applications.

15.1 Installation

To use the Java interface, one has to ensure that the environment variables `BPDIR`, `CLASSPATH`, and `PATH` (Windows) or `LD_LIBRARY_PATH` (Solaris) are set correctly. For a Windows PC, add the following settings to `autoexec.bat`:

```
set BPDIR=c:\BProlog
set PATH=%BPDIR%;%PATH%
set classpath=.;%BPDIR%\plc.jar
```

and for a Solaris or Linux machine, add the following settings to `.cshrc`.

```
set BPDIR=$HOME/BProlog
set LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$BPDIR
set CLASSPATH=.:$BPDIR/plc.jar
```

The environment variables must be properly set. The archive file `plc.jar` in the directory `$BPDIR` (or `%BPDIR%`) stores the byte code for the class `bprolog.plc.Plc` that implements the Java interface, and the file `libbp.so` (`bp.dll`) in the same directory is a dynamic link file for B-Prolog's emulator.

15.2 Data conversion between Java and B-Prolog

The following table converts data from Java to Prolog:

Java	Prolog
Integer	integer
Double	real
Long	integer
BigInteger	integer
String	string (list of integers)
Object array	list
Object	$addr(I1,I2)

Notice that no primitive data type can be converted into Prolog. Data conversion from Prolog to Java follows the same protocol but a string is converted to an array of Integers rather than a String, and a Prolog atom is converted to a Java String.

Prolog	Java
integer	Integer
real	Double
atom	String
string	Object array
list	Object array
structure	Object

The conversion between arrays and lists needs further explanation. A Java array of some type is converted into a list of elements of the corresponding converted type. For instance, an `Integer` array is converted into a list of integers. In contrast, a Prolog list, whatever type whose elements is, is converted into an array of `Object` type. When an array element is used as a specific type, it must be casted to that type.

15.3 Calling Prolog from Java

A Prolog call is an instance of the class bprolog.plc.Plc. It is convenient to import the class first:

```
import bprolog.plc.Plc;
```

The class `Plc` contains the following constructor and methods:

- `public Plc(String functor, Object args[])` It constructs a prolog call where functor is the predicate name, and `args` is the sequence of arguments of the call. If a call does not carry any argument, then just give the second argument an empty array new Object[] .

- `public static void startPlc(String args[])` Initialize the B-Prolog emulator, where `args` are parameter-value pairs given to B-Prolog. Possible parameter-value pairs include:

`"-b" TRAIL`	words allocated to the trail stack
`"-s" STACK`	words allocated to the local and the heap
`"-p" PAREA`	words allocated to the program code area
`"-t" TABLE`	words allocated to the table area

  where `TRAIL`, `STACK`, `PAREA` and `TABLE` must all be strings of integers. After the B-Prolog emulator is initialized, it will be waiting for calls from Java. Initialization needs to be done only once. Further calls to `startPlc` have no effect at all.

- `public static native boolean exec(String command)` Execute a Prolog call as represented by the string `command`. This method is static, and thus can be executed without creating any `Plc` object. To call a predicate in a file, say `xxx.pl`, it is necessary to first have the Prolog program loaded into the system. To do so, just execute the method `exec("load(xxx)")` or `exec("consult(xxx)")`.

- `public boolean call()` Execute the Prolog call as represented by the `Plc` object that owns this method. The return value is true if the Prolog call succeeds or false if the call fails.

15.4 Calling Java from Prolog

The following built-ins are available for calling Java methods or setting fields of a Java object. The exception `java_exception(Goal)` is raised if the Java method or field does not exist, or if the Java method throws any exception.

- `javaMethod(ClassOrInstance, Method, Return)` Invoke a Java method, where

  - `ClassOrInstance` is either an atom that represents a Java class's name, or a term `$addr(I1,I2)` that represents a Java object. Java objects are passed to Prolog from Java . It is meaningless to construct an object term by any other means.
  - `Method`: is an atom or a structure in the form `f(t1,...,tn)` where `f` is the method name, and `t1,...,tn` are arguments.
  - `Return`: is a variable that will be bound to the returned object by the method.

- `javaMethod(ClassOrInstance, Method)` The same as `javeMethod/3` but does not require a return value.

- `javaGetField(ClassOrInstance, Field, Value)` Get the value of Field of ClassOrInstance and bind it to Value. A field must be an atom.

- `javaSetField(ClassOrInstance, Field, Value)` Set Field of ClassOrInstance to be Value.

Chapter 16

Interface with Operating Systems

16.1 Building standalone applications

A standalone application is a program that can be executed without the need to start the B-Prolog system. You do not have to use the external language interface to build standalone applications. The first predicate that the B-Prolog interpreter executes is called `main/0`. To build a Prolog program as a standalone application, you only need to re-define the `main/0` predicate. The following definition is recommended[1]:

```
main:-
    get_main_args(L),
    call_your_program(L).
```

where `get_main_args(L)` fetches the command line arguments as a list of atoms, and `call_your_program(L)` starts your program. If the program does not need the command line arguments, then the call `get_main_args(L)` can be omitted.

The second thing you need to do is to compile the program and let your `main/0` predicate overwrite the existing one in the system. Assume the compiled program is named `myprog.out`. To let the system execute `main/0` in `myprog.out` instead of the one in the system, you need to either add `myprog.out` into the command line in the shell script `bp` (`bp.bat` for Windows) or start the system with `myprog.out` as an argument of the command line as in the following:

```
bp myprog.out
```

For example, assume `call_your_program(L)` only prints out `L`, then the command

```
bp myprog.out a b c
```

gives the following output:

```
[a,b,c]
```

[1]To use the graphics package CGLIB, you have to use the command `bpp`.

16.2 Commands

- `system(Command)` Send `Command` to the OS.
- `system(Command,Status)` Send `Command` to the OS and bind `Status` to the status returned from the OS.
- `chdir(Atom)`
- `cd(Atom)` Change the current working directory to `Atom`.
- `date(Y,M,D)` The current date is `Y` year, `M` month, and `D` day.
- `date(Date)` Assume the current date is `Y` year, `M` month, and `D` day. Then `Date` is unified with the term `date(Y,M,D)`.
- `time(H,M,S)` The current time is `H` hour, `M` minute, and `S` second.
- `time(Time)` Assume the current time is `H` hour, `M` minute, and `S` second. Then `Time` is unified with the term `time(H,M,S)`.
- `environ(EVar,EValue)` The environment variable `EVar` has the value `EValue`.

Chapter 17

Tabling

The need to extend Prolog to narrow the gap between declarative and procedural reading of programs has been urged long before. Tabling in Prolog is a technique that can get rid of infinite loops for execution of Prolog programs. With tabling, Prolog becomes more friendly to beginners and professional programmers alike. Tabling can alleviate their burden to cure infinite loops and redundant computations. Consider the following example:

```
reach(X,Y):-edge(X,Y).
reach(X,Y):-reach(X,Z),edge(Z,Y).
```

where the predicate `edge` defines a relation and `reach` defines the transitive closure of the relation. Without tabling, a query like `reach(X,Y)` would fall into an infinite loop. Consider another example:

```
fib(0, 1).
fib(1, 1).
fib(N,F):-
    N1 is N-1,
    N2 is N-2,
    fib(N1,F1),
    fib(N2,F2),
    F is F1+F2.
```

A query `fib(N,X)`, where `N` is an integer, will not fall into an infinite loop, but will spawn 2^N calls, many of which are variants.

The main idea of tabling is to memorize the answers to some calls, called *tabled calls*, and use the answers to resolve subsequent variant calls. In B-Prolog, tabled predicates are declared explicitly by declarations in the following form:

```
:-table P1/N1,...,Pk/Nk
```

where each `Pi/Ni` (i=1,...,k) is a predicate symbol and `Ni` is an integer that denotes the arity of `Pi`. To declare all the predicates in a Program as tabled, add the following line to the beginning of the program:

```
:-auto_table.
```

B-Prolog employs *linear tabling* which relies on iterative computation rather than suspension to compute fixpoints. A significant difference between linear tabling and suspension-based methods such as OLDT lies in the handling of variant descendents of a subgoal. In linear tabling, after a descendent consumes all the answers, it fails. A subgoal is called a *looping subgoal* if a variant occurs as a descendent in its evaluation. The evaluation of looping subgoals must be iterated to ensure the completeness of evaluation.

A data area, called `table area`, is used to store tabled calls and their answers. The following predicate initializes the table area.

```
initialize_table
```

The table area is initialized automatically in the beginning of each top-level query.

Tabled calls are stored in a hashtable, called `subgoal table`, and for each tabled call and its variants, a hashtable, called *answer table*, is used to store the answers for the call. The bucket size for the subgoal table is initialized to `9991`. To change or access this size, use the following built-in predicate:

```
subgoal_table_size(SubgoalTableSize)
```

which sets the size if `SubgoalTableSize` is an integer and gets the current size if `SubgoalTableSize` is a variable.

The following two built-ins are provided for fetching answers from the table.

- `table_find_one(Call)` If there is a subgoal in the subgoal table that is a variant of `Call` and that has answers, then `Call` is unified with the first answer. The built-in fails if there is no variant subgoal in the table or there is no answer available.

- `table_find_all(Call,Answers)` `Answers` is a list of answers of the subgoals that are subsumed by `Call`. For example, the `table_find_one(_,Answers)` fetches all the answers in the table since any subgoal is subsumed by the anonymous variable.

Chapter 18

Profiling

18.1 Statistics

The predicates `statistics/0` and `statistics/2` are useful for obtaining statistics of the system, e.g., how much space or time has been consumed and how much space is left.

- `statistics` This predicate displays the number of bytes allocated to each data area and the number of bytes already in use. The output looks like:

```
Control stack + Heap: 4000000 bytes
Control stack in use: 32 bytes
Heap stack in use:    200 bytes

Program area:         4000000 bytes
Program area in use:  482984 bytes

Trail stack:          2000000 bytes
Trail stack in use:   92 bytes

Table area:           2000000 bytes
Table area in use:    1324 bytes

Number of GC calls:   0
```

- `statistics(Key,Value)` The statistics concerning `Key` are `Value`. This predicate gives multiple solutions on backtracking. The following shows the output the system displays after receiving the query `statistics(Key,Value)`.

```
| ?- statistics(Key,Value).

Key = runtime
Value = [633,633]?;
```

```
Key = program
Value = [483064,3516936]?;

Key = heap
Value = [364,3999604]?;

Key = control
Value = [32,3999604]?;

Key = trail
Value = [108,1999892]?;

Key = table
Value = [1324,1998676]?;

key = gc
Value = 0?;
no
```

The values for all keys are lists of two elements. For `runtime`, the first element denotes the amount of time in milliseconds elapsed since the start of Prolog and the second element denotes the amount of time elapsed since the previous call to `statistics/2` was executed. For the key `gc`, the number indicates the number of times the garbage collector has been invoked since B-Prolog was started. For all other keys, the first element denotes the size of memory in use and the second element denotes the size of memory still available in the corresponding data area.

- `cputime(T)` The current cpu time is `T`. It is implemented as follows:

```
cputime(T):-statistics(runtime,[T|_]).
```

18.2 Profile programs

The source program profiler analyzes source Prolog programs and reports the following information about the programs:

- What predicates are defined?
- What predicates are used but not defined?
- What predicates are defined but not used?
- What kinds of built-ins are used?

To use the profiler, type

```
profile_src(F)
```

or

```
profile_src([F1,...,Fn])
```

where `F` and `F1,...,Fn` are file names of the source programs.

18.3 Profile program executions

The execution profiler counts the number of times each predicate is called in execution. This profiler is helpful for identifying which portion of predicates are most frequently executed.

To gauge the execution of a program, follow the following steps:

1. `profile_consult(File)` or `profile_compile(File),load(File)`. The program will be loaded into the system with gauging calls and predicates inserted.
2. `init_profile`. Initialize the counters.
3. Execute a goal.
4. `profile`. Report the results.

18.4 More statistics

Sometimes we want to know how much memory space is consumed at the peak time. To obtain this kind of information, one needs to recompile the emulator with the definition of variable `ToamProfile` in `toam.h`. There is a counter for each stack and the emulator updates the counters each time an instruction is executed. To print the counters, use the predicate

```
print_counters
```

and to initialize the counters use the predicate

```
start_click
```

Chapter 19

Frequently Asked Questions

How can I get rid of the warnings on singleton variables?

Typos are in most cases singleton variables. The compiler reports singleton variables to help the user detect typos. You can set the Prolog flag `singleton` to off to get rid of the warnings.

```
set_prolog_flag(singleton,off)
```

A better way to get rid of the warnings is to rename singleton variables such that they all start with the underscore _.

How can I deal with stack overflows?

The system does not enlarge a stack when it overflows. You can specify the amount of space allocated to a stack when you start the system. For example,

```
bp -s 4000000
```

allocates 4 mega words, i.e., 16 mega bytes, to the control stack. You can use the parameter '-b' to specify the amount allocated to the trail stack, '-p' to the program area, and '-t' to the table area. See 9.1 for the details.

Is it possible to set break points in the debugger?

Yes. Break points are also called spy points. You can use `spy(F/N)` to set a spy point and `nospy(F/N)` to remove a spy point. You can control the debugger and let it display only calls of spy points. See 6 for the details.

Is it possible to debug compiled code?

No, debugging of compiled code is not supported. In order to trace the execution of a program, you have to consult the program. Consulted programs are much slower and consume much more space than their compiled code. If your program is big, you may have to split your program into several files and consult only the ones you want to debug.

I have a predicate defined in two different files. Why is the definition in the first file still used even after the second file is loaded?

When a program in a file is compiled, calls of the predicates that are defined in the same file are translated into jump instructions for the sake of efficiency. Therefore, even if new definitions are loaded, the predicates in the first file will continue to use the old definitions unless the predicates themselves are also overwritten.

How can I build standalone applications?

You can use the external language interface with C or Java to make your program standalone. You can also make your program standalone without using the interface. You only need to redefine the `main/0` predicate, which is the first predicate executed by the B-Prolog interpreter. See Section 16.1 for the details.

How can I disable the garbage collector?

Set the Prolog flag `gc` to `off` as follows: `set_prolog_flag(gc,off)`.

Why do I get the error message when I compile a Java program that imports bprolog.plc.Plc?

You have to make sure the environment variable `classpath` is set correctly. Add the following setting to `autoexec.bat` on Windows,

```
      set classpath=.;%BPDIR%\plc.jar
```

and add the following line to `.cshrc` on Unix,

```
      set classpath=.:$BPDIR\plc.jar
```

In this way, `classpath` will be set automatically every time when your computer starts.

Can I have a Prolog variable passed to a Java method and let the Java method instantiate the variable?

No, no Prolog variable can be passed to a Java method. You should have the Java method return a value and have your Prolog program instantiate the variable. If you want a Java method to return multiple values, you should let the Java method store the values in the member variables of the enclosing object and let Prolog to use `javaGetField` to get the values.

Is it possible for one language to know about exceptions raised by a different language?

A call to a C function raises an exception if the function returns `BP_ERROR`. The global C variable `exception` tells the type of the exception. The exception can be caught by an ancestor catcher just like any exceptions raised by built-ins. The call `java_method` throws `java_exception(Goal)` if the Java method is not defined or the Java method throws some exception. The exception `java_exception(Goal)` can also be caught by an ancestor catcher in Prolog.

The C function `initialize_bprolog` returns `BP_ERROR` if the B-Prolog system cannot be initialized, e.g., the environment variable `BPDIR` is not set. The C functions `bp_call_string`, `bp_call_term`, and `bp_next_solution` return `BP_ERROR` if any exception is raised by the Prolog program.

In the current version of JIPL, the methods `Plc.exec` and `Plc.call` returns `boolean` and thus cannot tell whether or not an exception has occurred in the Prolog execution. The users' program must take the responsibility to inform Java of any exceptions raised in Prolog. To do so, the Prolog program should catch all exceptions and set appropriate member variables of the Java object that started the Prolog program. After `Plc.exec` or `Plc.call` returns, the Java program can check the member variables to see whether exceptions have occurred.

Is it possible to write CGI scripts in B-Prolog

Because of the availability of the interfaces with C and Java, everything that can be done in C, C++ or Java can be done in B-Prolog. So the answer to the question is yes. B-Prolog, however, does not provide special primitives for retrieving forms and sending html documents to browsers. The interface of your CGI scripts with the browser must be written in C or Java.

Chapter 20

Predefined Operators

```
op(1200,xfx,[:-, -->]).
op(1200,fx,[?-, :-]).
op(1198,xfx,::-).
op(1150,xfy,[?, :]).
op(1150,fy,[public,mode,dynamic]).
op(1100,xfy,;).
op(1050,xfy,->).
op(1000,xfy,’,’).
op(900,fy,[spy, not, nospy, \+]).
op(760,yfx,[#<=>]).
op(750,yfx,[#=>]).
op(740,yfx,[#\/]).
op(730,yfx,[#\]).
op(720,yfx,[#/\]).
op(710,fy,[#\]).
op(700,xfx,[is, in, \==, \=, @=, @:=, @>=, @>, @=<, @<, ?=, >=, >, =\=, ==,
            =<, =:=, =.., =, <, #>=, #>, #=<, #=, #:=, #<,#\=]).
op(661,xfy,.).
op(500,yfx,[\/, /\, -, +, .., \]).
op(500,fx,[-, +]).
op(400,yfx,[>>, <<, //, /, />, /<, *]).
op(400,xfx,mod).
op(300,xfx,**).
op(200,yfx,^).
```

Index

Index of Keywords

Index of Built-ins